The Anatomy of a Compiler

VNR

VAN NOSTRAND REINHOLD COMPANY

NEW YORK CINCINNATI TORONTO LONDON MELBOURNE

The Anatomy of a Compiler

JOHN A. N. LEE

Head, Computer Science Program,
University of Massachusetts,
Amherst, Massachusetts

DEDICATED TO:

My Colleagues of the International Federation of FORTRAN Fiddlers (an informal division of the COMMON User's Group) and in particular to the Canadian Programming Team

VAN NOSTRAND REINHOLD COMPANY Regional Offices:
New York Cincinnati Chicago Millbrae Dallas

VAN NOSTRAND REINHOLD COMPANY Foreign Offices:
London Toronto Melbourne

Library of Congress Catalog Card Number: 67-29207

Manufactured in the United States of America

Published by VAN NOSTRAND REINHOLD COMPANY
450 West 33rd Street, New York, N.Y. 10001

15 14 13 12 11 10 9 8 7 6 5 4 3

Illustrated by Karin Dekoj

Preface

The computer could not have progressed past the stage of being merely the plaything of the scientist if it had not been for Von Neumann's stored program concept and the consequent emancipation of the programmer from the toil of machine language programming by the development of higher level languages. In one short decade the science of computing has moved from the mystic world of backroom boffery to an ivory tower of free thought, wherein no one prospective user is forbidden to think or speak in the same monotones as the computer. Yet, with all the effort expended on the part of the manufacturers to produce languages which are easier to learn and use, the virulent and verbose user has always demanded more, so that we are now tending to develop more complex languages, acceptable to the old hand but incomprehensible to the newcomer.

Similarly, although the family of translators has been in existence for a considerable time (with respect to the age of the computer), the techniques of translation have been hidden in the tomes of technical obscurity. Ingerman,[†] in his own preface, commented that the available literature was "vague, obfuscatory and hubistic."

This text attempts to lift a portion of the veil of secrecy from compiler writing, though by no means can it be considered a "state of the art" manual. Rather, the author intends to show that, with some thought, the task of compiler writing is little more than that required to write an oversize program for the solution of a problem in a field foreign to the implementer. In fact, since the art of compiler writing is not yet a subject for study in a liberal arts course, it may forever remain the property of the specialized few.

[†] P. Z. Ingerman, *A Syntax Oriented Translator,* Academic Press, New York, 1966.

Yet today, the specialized few are the self-taught masters of the computing fraternity, who, recognizing their own shortcomings, continually grasp at the yet unobtainable. Using this book as the text for a course in compiler writing, it is the author's hope to short circuit the learning process of the prospective compiler writer and so to give him the opportunity, far earlier in his career than his predecessors, to beat against the barriers of our self-accrued knowledge and to extend the capabilities of the computer to newer linguistic levels.

There has been a tendency over the past few years to formalize the "theory" of compiler design. In particular, there has been developed a field of study concerned with the formal definition of computer languages both with respect to the syntax of the languages and the semantics of the source languages. Alongside these developments there has grown a set of programs which will accept these formal definitions as descriptors of a language, so that any source language may be converted to any desired target language. However, although formal syntax rules have been accepted as a standard means of language definition, the semantic definitions of language are still subject to criticism as being either machine dependent (even if the machine is hypothetical) or not susceptible to machine input. Further, since semantic definition must eventually describe the relationship between the source language and the target language, it is necessary that the author of the semantic definition be thoroughly conversant with the techniques of representing the procedures of the source language in the target language.

Based on these observations, this text describes the formal definition of syntax and the consequent syntactical analysis of input strings as well as the techniques of syntactical analysis which are peculiar to one language. Formal semantic definition has been omitted since no standard acceptable method of representation exists. Instead, the various compiler generators and routines are described, relating the source language to the target language in a less formal manner.

For the purposes of example, FORTRAN has been chosen as the basis for discussion. Each chapter describing the generators also discusses some extensions to the FORTRAN language which can easily be added to most systems and which incorporate many of the special features of other languages. The author regards these embellishments of FORTRAN as natural extensions of the language and not frills added for the sake of "frill mongering."

Acknowledgments

The implementation of a compiler such as that described herein, rarely is the work of a single person. Similarly, this text is truly a summary of the concepts, ideas (both accepted and rejected) and inspirations of a group of compiler writers. To my colleagues in the KINGSTRAN development group, I wish to express my deepest thanks:

Prof. J. Field, University of Waterloo, Ontario, Canada,
Dr. D. A. Jardine, DuPont of Canada Ltd., Kingston, Ontario, Canada,
Dr. E. S. Lee, University of Toronto, Ontario, Canada,
and Dr. D. J. Robinson, DuPont of Canada Ltd., Kingston, Ontario, Canada.

I also wish to thank those students in courses at the University of Massachusetts who used the early manuscripts as a text and who made many suggestions and corrections. Appendix D is mainly of their making. Also, to those students in the Computer Science Program at the University of Massachusetts who have successfully implemented compilers based on the concepts presented herein, go my heartfelt thanks for the encouragement engendered by their success.

For suggestions for revision and pointing out errors in the original editions, I wish to express my thanks to

Dr. Sue N. Stidham, University of Massachusetts, Amherst, Mass.,
and Mr. J. Stansburg, Halcon International, New York.

Amherst, Mass., Nov. 1968 John A. N. Lee

Contents

Appendices

1

Introduction

To the novice, kneeling at the footstool of the all-powerful programming instructor, the computer is sufficiently complex that he cannot begin to understand the even greater complexities of the compiler. Even yet, the experienced programmer, having written his program in some high-level language, goes to bed each night, satisfied that both the computer and the compiler performed their tasks with an exactitude that could not be expected of him. He does not stop to wonder at the products of many man-years of effort that have culminated in his ability to do his own work devoid of the frustrations of bits, bytes and binits. Yet let him take care, for even if one electron fails to find its right path home, he will curse every man-minute until he is proven exemplary.

Confronted with another machine of our age, the automobile, his personality undergoes a radical change, for having been exposed to the operation of an internal combustion engine, he is sensitive to every rattle. At the first sign of trouble, he confronts the garage mechanic with a barrage of constructive suggestions as to the probable cause of the trouble, knowing full well that given the time, tools and parts, he could fix it himself.

In the same manner, this text will allow the programmer to move from the stage of merely being the "driver" to a state of understanding where he can be a diagnostician, a fiddler or even an implementer. To this end, let us therefore review some of the concepts in computer language translation.

Translators

The mention of computer translation to the noncomputer person often leads to the thought of natural language translation, whereas, in fact, most of the work performed within a computer is a form of translation. That is, if we will accept the definitions †:

> *translate, v.t., ... to transform; to render into another language; to explain by using other words; . . . .*

as being applicable in this context, then the transformation of data by an algorismic process can be considered a translatory process. Further, since Hamming ‡ has decreed that:

> *"The purpose of computing is insight, not numbers"*

then the computer is an explanation machine. However, by general usage in "computerese," *translator* is the generic term for computer programs that accept as input a nonnatural computer language and output some other nonnatural computer language. Only when modified by certain adjectives, may the term "translator" refer to any more specific system. Thus we shall encounter such terms as "syntax-oriented translator" and "natural language translator."

In one sense, a cryptographer may be considered a translator; however, the difference between creating a cipher from a message given the key (algorithm) and decoding a cipher to a message lacking the key is extremely great. As an example of the latter, consider the task undertaken by Legrand, the leading character in Edgar Allan Poe's *The Gold Bug,* who discovers the following cipher:

† From the *Webster Modern Reference Dictionary of the English Language,* Consolidated Book Publ., Chicago, 1964.

‡ R. Hamming, *Numerical Analysis for Engineers and Scientists,* McGraw-Hill Book Company, New York, 1964.

```
53◆◆†305))6*;4826)4◆.)4◆);806*;48†8¶60))
85;(;}8*);:◆*8†83(88)5*†;46(;88*96*?;8)*◆(;485);
5*†2:*◆(;4956*2(5*-4)8¶8*;4069285);)6†8)
4◆◆;1(◆9;48081;8:8◆1;48†85;4)485†528806*81(
◆9;48;(88;4(◆?34;48)4◆;161;:188;◆?;
```

Having decided by other logical reasoning that the author of the cipher was concealing a message originally written in English, and recognizing that there were no word delimiters, Legrand made a count of the frequency of occurrence of the individual characters:

Character	*Number of Occurrences*
8	33
;	26
4	19
◆	16
)	16
*	13
5	12
6	11
†	8
1	8
0	6
9	5
2	5
:	4
3	4
?	3
¶	2
}	1
-	1
.	1

Now the frequency of occurrence of letters in the English language is (from highest to lowest):

eaoidhnrstuycfgvlmwbkpqjxz

Observing that the accretion *ee* occurs often and that a common three character group is *the,* Legrand comes to the conclusion that 8 represents *e* , ; represents *t* and 4 stands for *h.* After much further rumination based on the possible character combinations in the English language (such as the fact that the accretion *qq* never occurs), the following partial key is developed:

Character in Cipher	*Character in Message*
5	*a*
†	*d*
8	*e*
3	*g*
4	*h*
6	*i*
*	*n*
◆	*o*
(	*r*
;	*t*

Legrand then inserts the other characters in the cipher after further inspection and produces an unpunctuated version of the message. Only by close examination of the actual script and a knowledge of the neighborhood, does he determine the actual message:

> *A good glass in the Bishop's hostel in the Devil's seat—twenty-one degrees and thirteen minutes—northeast and by north—main branch seventh limb east side—shoot through the left eye of the death's head —a bee line from the tree through the shot fifty feet out.*

While Sherlock Holmes was no less adept at cryptography[†] than Legrand, he was an expert at deduction to the point that the amalgamation of data lead him to some remarkable conclusions. Even this process, in a sense, can be considered translation, since it is an interpretation of data leading toward a target, that is, an explanation using different words.

Computer Languages

Since the first Von Neumann computer was built with the essential feature of a stored program, the problem of communication between man and machine has been ever present. While the concept of program writing instead of wiring a plug board seemed, at the time, to be the ultimate weapon of machine direction, machine language was not even close to being a final weapon of the computer fraternity when it became apparent that to learn machine language was equivalent to subjecting oneself to the rigors of any discipline. Thus only those persons who could spare the time or who were already involved in the industry made the effort to acquire this knowledge. From this beginning grew an elite band of experts who were capable of communicating with the computer.

However, the use of a middleman is always objectionable, and the person with the problem found that the task of describing the problem and answering the poorly defined subproblems was far more trouble than actually solving the problem by hand. Thus the development of problem-solving languages has been for the benefit of the noncomputer expert and not primarily for the professional programmer.

Continued language development by manufacturers in the competitive arena has not only meant the evolution of more meaningful communication systems, but also a diversity of languages that are machine- or, at least, manufacturer-dependent.

To complicate this situation, specialized languages that are slanted toward families of problems in certain disciplines have also developed. Thus the language useful to the civil engineer will be of little use and perhaps even meaningless to a physicist.

The so-called algorithmic or procedure-oriented languages, such as ALGOL, FORTRAN and PL/I, were intended to provide a mode of communication akin to normal (as opposed to natural) mathematical nomen-

[†] See *The Adventure of the Dancing Men* by Sir Arthur Conan Doyle.

clature. However, the restrictions imposed by these languages have confined their use to problems that are well defined in scalar algebra. This has, in some instances, led to the development of brilliant techniques to circumvent these shortcomings; however, the effort expended or required to create the algorisms, by those who are merely trying to solve a problem, has tended to create the illusion in some minds that the mystic art of programming is still a land of forbidden territories to the nonprogrammer.

One of the foremost difficulties we have experienced in the use of the first two generations of computers has been the fact that the computer could accept only a linearized input medium. Only in the third generation are we being supplied with input devices capable of accepting a two-dimensional communication. With the development of page-reading devices, enabling a communication with the computer in humanly readable form, it will be possible, eventually, to present a problem definition in terms of the algorithm printed in a standard text or technical journal, adding to this, for data, a sketch of the known information. In one instance [†] such a two-dimensional input system has been successful. Using a modified flexowriter as an input device, a page of standard text, complete with integrand and summation signs, was introduced into the computer and a linearized algebraic program was produced. Such a system is sure to be implemented with a cathode ray tube and light pen as input devices.

Besides the desire to manipulate numerical values, the need to manipulate algebraic and symbolic data is of significance. Since much of the work in the design of a new engineering product consists of the manipulation of algebraic expressions, with the subsequent substitution of numeric values for the algebraic symbols, it is reasonable to request that the computer undertake the menial and well-defined tasks of manipulation. Thus the development of algebraic manipulation languages is of considerable importance. To such an end primitive systems such as FORMAC [‡] and ALPAK [§] have been developed.

While both systems are excellent examples of the road to be pursued, the present restrictions lead the author to believe that by the time both

[†] M. Klerer, "Two Dimensional Programming," Proceedings of the Fall Joint Computer Conference, Las Vegas, 1965.

[‡] E. Bond et al., "On FORMAC Implementation," *IBM Systems Journal,* Vol. 5, No. 2, 1966. Samples of FORMAC input and output are given in Appendix B of this text.

[§] W. S. Brown, "The ALPAK System," *Bell System Technical Journal,* Vol. XLII, No. 5, 1963.

systems have been developed to a satisfactory level, other systems will have been produced that will preclude their general acceptance. For example, the logical decision-making facilities of both systems discourage the writing of extensive production programs and encourage their use for one-off problems. Further, in a time-sharing remote-access environment with cathode ray tube displays, an algebraic manipulator could fall back on the real-time user to define the tasks to be undertaken, thereby eliminating the necessity to prewrite a program.

Machine Language Coding

A programmer writing in the basic language of the computer can stay close enough to the elemental operating procedures of the computer to take advantage of some specialized techniques, but he has to contend with several major disadvantages. Among these we may list:

(a) All operation codes and operand addresses must be written in some numeric code.

(b) All addresses in this code must be absolutely defined. Thus the programmer must either possess extrasensory perception—enabling him to choose the address of a piece of data or instruction not yet defined—or be prepared to backtrack over the coding and fill in holes that were left when he attempted to make the forward references.

(c) Any changes in coding or data assignment (such as the insertion of an instruction) will necessitate the reassignment of many existing data and instructions, and the consequent modifications of all references in the original program.

(d) Though the actual data to be processed may be in (say) decimal mode, the programmer must (for most computers) convert this data to binary mode for manipulation.

(e) An inherent fear of the necessity to rearrange the addressing structure of the program often leads the programmer to leave sufficient space around certain portions of the program and data to permit insertions. This leads in turn to an inefficient use of the available memory. This disadvantage is not inherent to machine language coding, but rather to most machine language programmers.

(f) Since mere coding does not solve the original problem, the programmer is also faced with the tasks of transferring the coding into a machine-readable form, proofreading the transformation, and loading the code into the memory of the computer.

(g) Portions of previously written programs, which may have a use in the present coding effort, cannot be included without being recoded to conform with the present address assignments.

Symbolic Language

Because of the format of machine language and the consequent tediousness of transforming a problem definition into that language, *symbolic languages* have been developed to overcome many of the inconveniences. In conjunction with an assembler, the programmer uses mnemonic, easily remembered codes for operators, instead of bit patterns or octal codes, and can choose names to represent data items and the addresses of instructions. Further, representations of data are automatically converted to the internal machine mode.

The *assembler* is a machine language program that translates symbolic code into machine language and also provides the facilities to relieve the programmer of many housekeeping chores. For example, since symbolic language is generally a series of statements with a one to one correspondence with machine language instructions, it is a simple task for the assembler —given a starting address for the first instruction—to keep track of the relationship between symbolic instruction names and actual machine addresses. Thus symbolic references to instructions can be correctly related to the referenced instruction.

To ease the task of coding, a symbolic code not only has statements that are in a one to one correspondence with machine codes, but also *declaratives,* which enable the programmer to declare constants and data areas and to append symbolic names thereto, and *control statements,* which will direct actions within the assembler itself.

An assembly system may also contain the ability to incorporate sets of standard instructions by including a single statement in the symbolic code. In certain systems, all references produce linkages to such standard routines. Such a system is a *macroassembler.* For example, some computers have no built-in floating point hardware, whereas most scientific computations are performed in the floating point mode. Thus such a system, which enables the programmer to write a code akin to a machine language, but which in fact creates linkages between the data and the specialized routines that simulate floating point hardware, is an immense boon.

Interpreters

Within the collection of names given to members of the family of translators is that of *interpreter.* It would seem logical to assume that this would be a pseudonym for a translator, but again, a special meaning is engendered in the mind of the computeree by this word. In most cases, the program, which fulfills the assignment of being an interpreter, possesses many of the qualities of a compiler or an assembler, but manifests one important difference.

If we consider both the compiler and assembler to be translators, the target of each being a language readily understandable to the computer and by means of which the problem described in the original language can be executed, then each may be considered to be a two-phase system, with a translation phase and an execution phase. Moreover, each phase is intact, the execution phase being capable of being repeated without limit and without recourse to the translation phase as shown in Fig. 1.1.

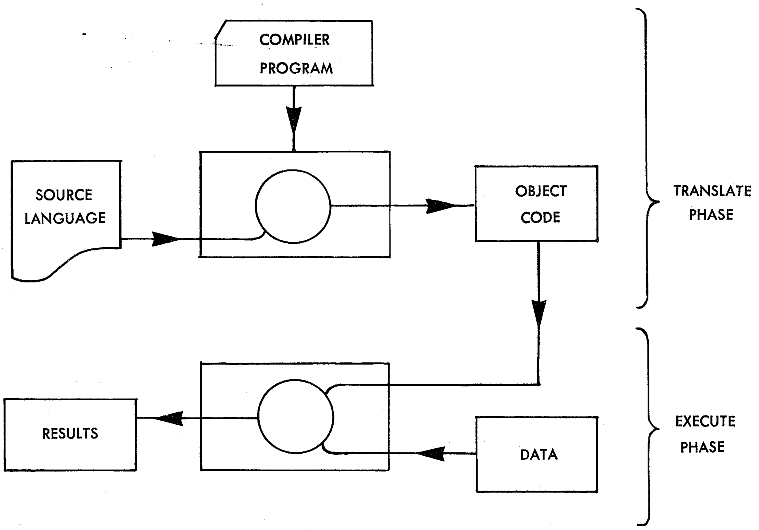


FIGURE 1.1

An interpreter, on the other hand, does not have distinct translate and execution phases (Fig. 1.2), the two being interleaved continually. That is, as soon as one phrase of the source language has been translated to an executable code, the code is executed without waiting for the translation of the complete source document. Further, since the interpreter must be resident in the computer memory during execution, and hence memory space is at a premium, the executable code is not saved. This then implies two other features:

(a) If looping is to be permitted in the problem description, the source language must also be retained in memory.

(b) Even though a phrase may have been translated once, any attempted reexecution of that phrase will require a retranslation.

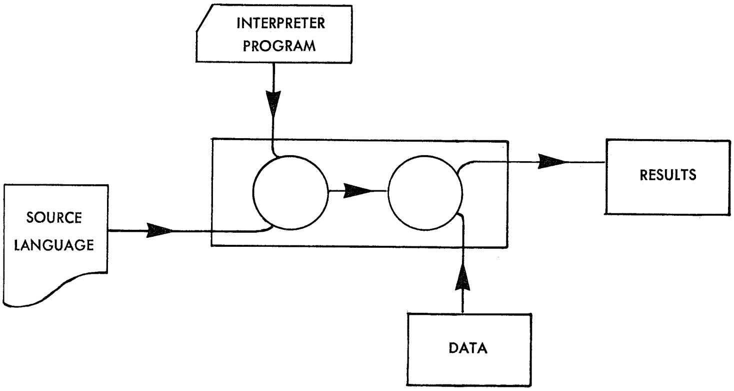


FIGURE 1.2

As a consequence, the execution speed of a program operating under the control of an interpreter is slow, and only relatively small programs can be executed. However, on the credit side, there is a definite affinity between the object code and the source language at all times, so that diagnostics during the execution phase can be stated in terms of the original code written by the programmer rather than in terms of the object code with which he may not be familiar.

Many interpretive systems have been stopgaps in providing systems for newer machines intended to replace existing computers; they are intended to save an overwhelming burden of reprogramming at the instant of the changeover. For example, when the author was Director of Computing at Queen's University at Kingston, Ontario, in 1961, the Computing Center converted from a Bendix G-15D to an IBM 1620. During the period in which the Bendix had been resident, many programs had been written in a language known as INTERCOM 500, itself an interpreter. Such a language was not immediately available on the IBM 1620 and thus, to ease the burden of reprogramming, a similar compatible language interpreter, QUICK,[†] was developed. With the use of this system, many already operating programs were operational immediately, and no manpower was lost among the users as a result of the changeover—though, of course, manpower was utilized in the development of the system.

However, such a system substantially slowed down the potential speed of the computer and subsequently an assembly system was written, EASY, [‡] which performed a simple translation from the INTERCOM language to the machine language of the IBM 1620. At this point most of the speed advantage of the newer machine had been achieved and already debugged programs from the G-15D were operational on the 1620. However, new programs were being written in the languages available for the 1620.

With the advent of System/360, interpreters are important again, taking their place in the panorama of programs to enable programs that were written in the machine language of older machines to operate on this new machine. However, this is not unexpected since such interpreters, known in this peculiar instance as *simulators,* were available on prior machines. What is different here is that the interpreter for System/360 is partially implemented by the provision of auxiliary computer hardware, and consequently has been given the name of *emulator.*

The eminent computer text author, D. D. McCracken, tells the story of a certain corporation which, in the infancy of the computer business, possessed a machine known as the CPC, for which it had many programs written, including one for payroll calculations. The CPC was a pre-Von Neumann machine, which had no stored program and which interpreted instructions punched into a sequence of cards to activate the execution of the program. Looping was achieved by the operator who took the instruction cards from the output stacker and returned them to the input hopper.

Eventually, the CPC was replaced by an IBM 650, a far superior machine, but with an incompatible machine language. Thus to save reprogramming, a CPC simulator was written and existing programs were operated under this program's control. When the 650 was replaced by the

[†] J. A. N. Lee, "Queen's University Interpretive Coder, Kingston." Report No. 28, Queen's University Computing Centre, 1962.

[‡] J. A. N. Lee, "Exchange Assembly System," Report No. 30, Queen's University Computing Centre, 1962.

IBM 704, it was only natural to write a 650 simulator for the 704 so that the 650 machine language programs could continue to operate without being reprogrammed. Of course, one of the programs to be run under the 650 simulator was the CPC simulator, under which the payroll program was still running.

The IBM 704 was eventually replaced by an IBM 7090 with a 704 simulator, and that machine by the STRETCH. . . .

Compilers

The differentiation of a compiler from the family of translators is not as well defined as the difference between other types of translators. While we may clearly classify an assembler as a program that translates from a symbolic language to an executable code in essentially a one for one process, and an interpreter as an interleaved translator and executor, the compiler may be regarded as a pure translator in which the source language is machine independent and the object code machine dependent.

Further, a compiler may be considered as two basic parts together with a set of utility routines. The first part is an analyzer, or sieve, which has the ability to distinguish between the types of statements in the source language. The second part is a set of generators that produce the object code on the basis of the relevant information in each source language statement. The choice of generator to be used for a particular statement type is the task of the sieve.

In many compilers, most of the memory taken by the system is used up by the utility routines, the arithmetic scanner and its associated generator. The other generators are then relatively small routines, each extracting from the source document the relevant information to be substituted into the object code by referring to the utility routines. For example, the symbol and number extraction routines are used by almost every generator. Similarly, the symbol table routine provides data on object-time-memory allocation to each generator on the basis of the symbols provided by previous statements.

Once such a set of utility routines is available in a compiler, extensions to the language merely require the introduction of a new exit to the analyzer linking to a new generator. At this point, the problem of writing a new generator and sieve exit may be far less severe than that of acquiring sufficient computer time to reassemble the compiler.

A conceptual difficulty that must be overcome by the compiler writer, and one for which he must have a well-ordered mind, is that of keeping the various languages with which he is working separated. For example, it is conceivable that the compiler is being written in language A, which is really the symbolic code for machine language B, and the resulting compiler will accept language C for conversion to object code D.

It has been proposed[†] that the difference between a compiler and an interpreter may be defined in terms of the input and output data. The compiler accepts input in the form of a string of characters and, with reference to a set of production rules, produces another string of characters which, by pure coincidence, happen to be a set of machine language instructions to solve the problem defined in the input language. On the other hand, the interpreter uses the input strings as a set of instructions and by following these commands executes the algorithm, producing a set of output which is relevant to the solution of the problem.

Bootstrapping

Whenever a new computer is placed on the production line, its usability is not only dependent on the ingenuity of its electronic designers but also on the competence of the software development group. However, such a group, at its inception, is itself devoid of languages except the one built into the machine, that is, machine language. The group's first task then must be to produce an assembly system capable of producing machine code for the minimum set of instructions, that is, the set of instructions necessary to implement the minimum assembler. This recursive definitive of the minimum set of instructions is necessary for the next phase of development, for once the assembler has been written and debugged in machine language, it may be recoded in the assembly language itself. Thus after the initial writing, debugging and rewriting, the assembler may be reassembled from the assembler code.

Bootstrapping is a process of translation using language A as input to a program, written also in language A, to produce another translator that will accept language A+ as input. At the level of the assembler, bootstrapping is a common procedure for developing higher level assemblers. For example, in the primitive assembler there may be no facility for the declaration of data blocks, such blocks being declared by the repetitive definition of single

[†] D. Stemple, Systems Consultant, University of Massachusetts, Private communication.

words or the redefinition of the object-time-memory allocation register so as to leave an area of memory for storing the array. Similarly, the primitive system may not allow the occurrence of arithmetic operations within the symbolic operands, while this added feature in later versions will help ease program data referencing. Thus, as the implementation of an assembly system progresses, more convenience-features may be added. During such a development, it may be difficult to judge when one version is suitable for general release to the users and work on further embellishments should be halted. These are managerial decisions, each of which can substantially affect the subsequent success of the total computer system.

Though much rarer, there is a growing tendency on the part of both manufacturers and educational institutions to write the compiler for a procedure-oriented language in that language itself. There is at least one FORTRAN/FORTRAN and one ALGOL/ALGOL, while it is rumored that a high level PL/I is written in a lower level PL/I. One of the reasons for this emergence of the bootstrapped compiler is the need to teach compiler writing without the added complication of getting involved in the intricacies of machine code. In fact, most of the problems that appear in this text can be programmed in an algebraic language that also has the facility for the testing of alphabetic data, or any FORTRAN that allows Hollerith constants in assignment statements and IF statements.

At this level of sophistication it should be possible for a programmer, given the knowledge that the particular compiler available in the local computer shop is written in its own source language, to implement those features of the language which he feels to be omissions and to recompile the compiler. However, the haphazard intrusion of new features without regard for their interaction could lead to havoc in short order. As will be seen in later chapters, the avoidance of ambiguity in a language is of paramount importance in the design of the language itself; added features must not create a situation wherein a standard feature becomes unusable.

Problem-oriented Languages

The algebraic, algorithmic, or procedure-oriented languages such as ALGOL, FORTRAN, or PL/I are designed to further the communication of mathematical problems to the computer, and therefore may be considered to be special forms of problem-oriented languages. However, by general usage, problem-oriented languages refer to those languages that are restricted to the description of more specialized problems. For example, COGO[†] is a language for civil engineers and surveyors, and uses phrases containing keywords that are indigenous to the standard vocabulary of the prospective user. On the slightly lower level, a system known as MAGIC[‡] enables the user to describe computations in terms of matrix operations by using such key words as **ADD, INVERT,** or **SOLVE.**

Although most problem-oriented languages are interpretively translated and executed, this is more because of convenience than special design. However, it must be anticipated that such languages will be of importance in commercial time-sharing, remote-access systems, where a console may be placed in a small office where business is confined to a single type of product. Whereas in the past only large institutions and corporations could afford computers as the "jack-of-all-trades," the sharing of a central computer by many small companies will lead to the desire for the availability of more specialized languages at the console.

The major advantage of these problem-oriented languages lies in the fact that very few new notions are included above those learned by the prospective user in his own apprenticeship, and thus little time is lost in the programming course that elucidates the intricacies of the particular language. Using COGO, the author has found that a group of sophomore students taking a course in surveying (and their instructor), all of whom had no previous computer experience, could write competent programs after only one hour of instruction. In fact, most of these same students wrote a COGO program the same evening, which solved a surveying problem that was originally intended as a term project. Within 24 hours, the instructor revised his estimate of the work load in the course and was able to assign problems that required much more thought than those given previously, without the fear of an overload of tedious calculation.

If algebraic compilers can be regarded as one end of the spectrum of problem-oriented languages, then the compiler-compiler must be the other extreme. As opposed to the technique of bootstrapping, which has no real, distinct source language and which by definition is restricted to a single target language, a compiler-compiler is a translator that accepts a language descriptor as input, and outputs a compiler capable of translating the language described in the descriptor. In this manner a single descriptor language would be capable, along with a compiler-compiler, of producing

[†] D. Roos and C. L. Miller, "The Internal Structure of COGO," Report No. R64-5, Dept. of Civil Eng., Massachusetts Institute of Technology, February, 1964.

[‡] J. A. N. Lee, "MAGIC—A Matrix Algebra General Interpretive Coding," Report No. 43, Queen's University Computing Centre, 1964.

compilers for many different user languages. By this simple definition, any assembler might be considered to be a compiler-compiler, but since this is not the prime purpose of an assembler and since no special features to implement compiler writing are built into an assembler, we shall exclude assemblers from the class of compiler-compilers. Similarly, any system that is not specifically designed to compile compilers will be excluded from the set.

In 1967, there does not exist a single compiler-compiler that is available from any computer program library, though several exist as the proprietary programs of individual manufacturers. Similarly, although several have been discussed in the literature, there is no commercially available syntax-oriented translator.

Syntax-oriented Translator

As in a compiler, where the input consists of a language describing the problem to be solved and the output is an object code which, when executed, will solve the problem, so the syntax-oriented translator accepts a description of the language and produces † an object code that will translate such languages. See the schematic diagram in Fig. 1.3. Obviously, such a process involves a further language level, in that we now require a language to describe a language, and this entails another language to describe that language. At this stage we must resort to our own native tongue. Such language descriptors will be described in later chapters.

The approach proposed by the proponents of the syntax-oriented translator has the advantage of providing a system that will enable the experimenters and compiler writers to test their ideas; with a second descriptor to define the relationship between the language descriptor and the object code, the system could be machine independent. This means that a manufacturer could maintain a set of compiler language descriptors, including one for the syntax-oriented translator itself, and then by merely writing a single second descriptor for a new machine, he could make all systems immediately available. Similarly, the development of a new language by the writing of a new descriptor would make that language available on all machines that accept that descriptor. With any degree of competency, this might resolve the problem of a machine becoming outdated by the lack of up-to-date systems when the manufacturer moves his programming staff to

† It is uncertain, at the time of writing, whether a true syntax-oriented translator will actually produce an object program or will be self-modifying and thus become the compiler itself.

more modern computers. This, of course, will depend on the universal standardization of the language descriptors.

On the debit side, a compiler provided from a jack-of-all-trades translator cannot be as efficient as a compiler written specifically for the language. This fact, and the competency with which the gap between these two extremes is closed, may be the deciding factor as to whether such a system will be generally acceptable to the user, or whether the system will be relegated to the drawing board.

A specialized treatment of a syntax-oriented translator will be found in Ingerman.[†]

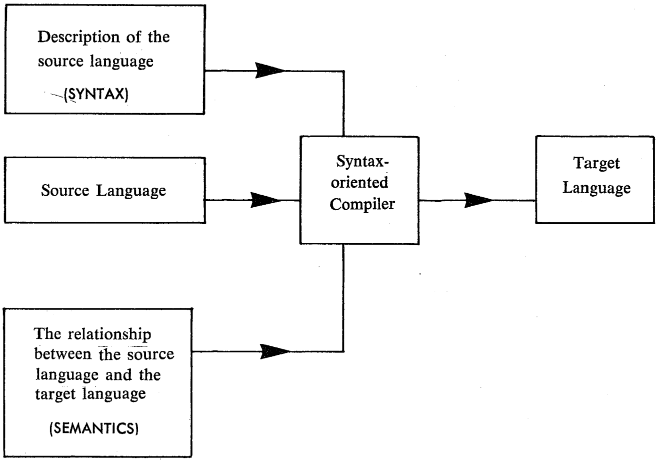

Description of the source language
(SYNTAX)
Source Language
Syntax-oriented Compiler
Target Language
The relationship between the source language and the target language
(SEMANTICS)

FIGURE 1.3 A Schematic of the Syntax-Oriented Translator

The Passes Quandary

In the implementation of a translatory process, one of the first decisions to be made is the number of passes to be included in the system. In a large computer system with high-speed, secondary level storage devices, the time to reload the main memory may not be significant, and therefore the overhead time in translating in phases with intermediate output to an auxiliary memory for input to a later phase can be accomplished at little cost. However, in a primitive computer system with slow auxiliary memory, or in one relying on intermediate card output and the input of phases from a card

[†] P. Z. Ingerman, *A Syntax Oriented Translator,* Academic Press, New York, 1966.

reader, the overhead time may be great enough to warrant minimizing the number of passes through the computer. Further, the availability of immediately accessible memory may decree that since the whole compiler or assembly system cannot reside in memory at one time, a multipass translation system is necessary.

If one defines the number of passes in a translatory process as the number of distinct programs that must be introduced into the computer to complete the conversion from source language to machine language, then it is evident that very few compilers are truly a single pass system. In fact, most compilers are four pass systems, as shown in Fig. 1.4. One of the

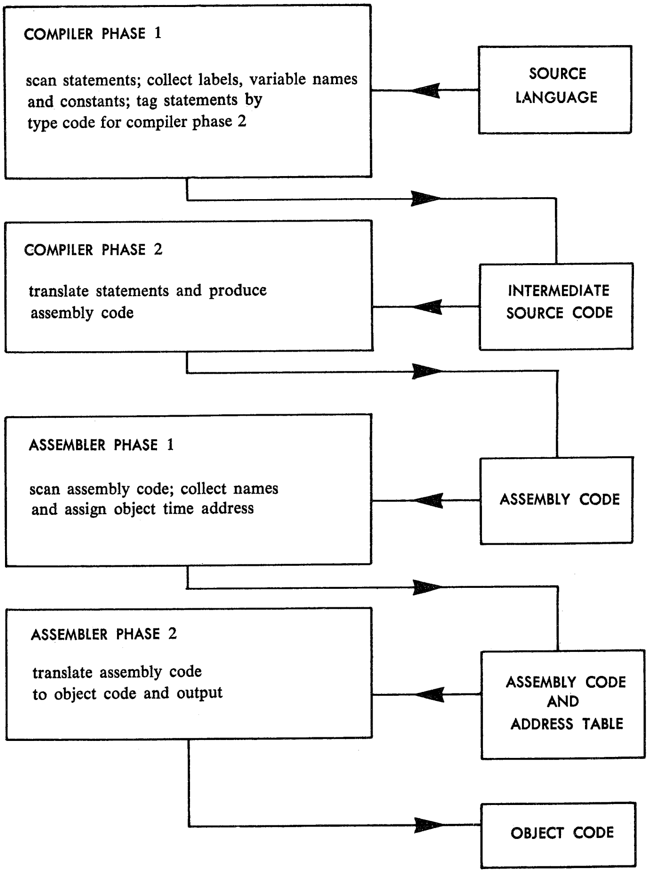

COMPILER PHASE 1
scan statements; collect labels, variable names and constants; tag statements by type code for compiler phase 2
SOURCE LANGUAGE
COMPILER PHASE 2
translate statements and produce assembly code
INTERMEDIATE SOURCE CODE
ASSEMBLER PHASE 1
scan assembly code; collect names and assign object time address
ASSEMBLY CODE
ASSEMBLER PHASE 2
translate assembly code to object code and output
ASSEMBLY CODE AND ADDRESS TABLE
OBJECT CODE

FIGURE 1.4 Typical Four-Pass Compiler Organization

principal advantages of such a four pass process is that it utilizes the utility routines in the assembler to allocate memory space, thus reducing the size of the basic compiler. Of secondary importance is the ability to obtain a copy of the intermediate assembly code. While this may be of considerable importance to some programming specialists, it is generally incomprehensible to the nonspecialist (who form 90% of the users of a computer), and only wastes machine time in its production. However, in a corporation where the task is to produce operating software in the shortest possible time, the duplication of effort in writing memory allocation routines for both the compiler and assembler is obviously not economical.

Thus if the features of the assembler are combined into the compiler, the number of passes can be reduced to two without loss of machine time or efficiency of the object code, although with the expenditure of memory to accommodate the extra code.

As is common in many physical processes, where it is possible to streamline the process and where the accomplishment of streamlining by a factor of two is comparatively straightforward although the further improvement takes an inordinate amount of energy, reducing the number of passes from two to one in a compiler causes innumerable headaches. A pseudo one-pass system can be accomplished by maintaining all processor phases in memory simultaneously and by storing intermediate results in the same memory, instead of using auxiliary storage devices. However, this is merely a degenerate multipass system, since it is not *required* that each phase exist in memory simultaneously. Thus a true one-pass system is one in which the omission of any one routine would not enable the whole scope of the language to be translated.

In general, the desire to implement a one-pass system will entail the adoption of one of the following solutions:

(a) The entire source program is stored in memory together with all phases of the compiler so that only one input and output operation is required. In this manner, the translator may scan each statement many times as if the processor were a multipass system.

(b) Stringent restrictions to the source language to include only those statements capable of being sustained in a one-pass system are applied.

(c) Relaxations in the target language efficiency are allowed, and some of the features existent in the program loader are utilized to overcome problems in backtracking.

The first of the alternatives presents no new ideas and merely decrees that, given sufficient memory, almost anything and everything can be accomplished. The second alternative detracts from the objective of implementing a translator for a proper language and does not allow for language extension. Thus we are left with the third alternative. However, it would be illogical to accept this as the best solution, having eliminated the other two, since it has not been established that only one alternative is best or that these are in fact the only three solutions. The acceptance or rejection of alternative (c) must be predicated on the answer to the question: *"How much efficiency do I lose?"* Also in any particular computer shop, the time lost in execution of the object code must be balanced against that gained during compilation. For example, in an educational institution where the greatest proportion of jobs are run for students in computer-programming courses, the time expended in compilation far exceeds that for execution, so that any decrease in compilation time is significant. On the other hand, in a production shop where a single program (debugged and compiled) may be run many times, the execution time is of considerable importance.

The major problems of compiling in a single pass are concerned with the compilation of forward references from one portion of the source program to yet undefined statements. In particular, a FORTRAN GO TO statement containing a statement identifier of a statement not yet encountered cannot be compiled as a simple unconditional jump or branch. This may be overcome by compiling an indirect branch referencing a memory word into which the address of the statement identified will be stored by the loader, or by outputting a branch instruction with a null address, which is to be overlaid by the actual address during the loading of the object program. The latter technique (known herein as backtracking) conserves object-time-memory space, but wastes both overhead time in the loader and temporary storage, which maintains a list of the addresses of the instructions to be overlaid. The former technique wastes only one object-time-storage location (per forward referenced identifier) and introduces extra machine cycles in chaining through the indirect addressing. These problems will be discussed in greater detail in Chapter 5.

In earlier generation computers, using drums as main memory with a 1+1 addressing system[†] in which each instruction contained the address of the next instruction to be executed, a forward reference could be coded directly and a notation left in a table designating the address at which

[†] For example, the Bendix G-15D and IBM 650.

the referenced statement was to start. The choice of the starting address of the referenced statement would be made on the basis of the time taken to interpret the branching instruction and the angular position of the drum at the time the next instruction was to be accessed. After the first forward reference to a particular statement identifier had been encountered, every other reference would be coded as if the identifier had been defined.

In a one-pass system, the interactive features of certain statements can cause insoluble problems if the ordering of statements is not specified. For example, the interaction of the FORTRAN statements DIMENSION, EQUIVALENCE and COMMON (as will be discussed in Chapter 4) can be overcome by requiring that the statements occur in the order:

COMMON
DIMENSION
EQUIVALENCE

and precede any executable statement in any subprogram.

While one group is striving to minimize the number of passes in a compiler, it is quite feasible that another group will elect to increase the number of passes so that a compiler can be implemented on a minimum-memory machine. For example, the users of a computer with minuscule memory have as much right to an algebraic compiler as the users of a larger machine. The means by which this right is attained may engender an entirely different philosophy from that for the implementation on a large machine, and —provided that the need justifies the computer time expended—the almost impossible can be brought about. Thus the building-block compiler was born.

As a starting premise, it has been found from experience that the amount of memory needed to store the source language in internal mode is greater than that required to accommodate the object code. Thus if a portion of the available memory is reserved for the storage of the source program, and the remainder is used for the compiler and work areas, it should be possible to convert the source code to object code in situ. This was found to be possible in the IBM 1401, but the memory remaining after the reservation of the input zone (which incidentally had to be large enough to contain certain object-time routines such as input/output and library functions) permitted only very small portions of the compiler to be resident at any one time. However, by placing all work areas in COMMON with respect to each phase of the compiler, a 61 pass system was developed to afford the opportunity of FORTRAN programming to 1401 users. A complete description of each phase is given in Appendix A.

The question *"how many passes"* may well be significant in the development of a time-sharing operation. In a computer system with a single central processing unit, the overhead time of moving data in and out of the main memory and searching for a specific compiler, assembler or utility system must be kept to a minimum, or else the setup times will accrue to the extent that the amount of actual production work is reduced to the point where time sharing is no longer economical. In a system with multiple processors and overlapped input/output, the location and transfer of shorter programs can be achieved with little loss of main processor time. Thus in a small time-sharing environment, a resident one-pass system may be more profitable (but restrictive on the size of program to be executed and the diversity of languages available), while in a multiprocessing situation, a building-block compiler with the ability to select the appropriate generator will be more suitable. Similarly, to gain a spectrum of languages at the console, a syntax-oriented translator resident in memory with shuffling descriptors could offer sufficient advantages to outweigh the disadvantages of the loss of memory and slower compilation and execution times.

Summary

The preceding review is certain to have omitted some of the concepts of some of the members of the computer fraternity, and to have propounded some definitions that will not satisfy all readers. However, neither can the following chapters be construed as representing all the techniques of any single system, each particular algorithm having a place in most systems. It is left to the reader therefore to construct his own translator for the purpose he desires, and not to be constrained by the artificial barriers of the semantic definitions of the terms compiler, assembler, etc.

The remainder of this text will describe the various routines and the algorithms that are used in compilers and assemblers to translate from the source language to the object code. Throughout, the object language shall be referred to as a symbolic code, though a one to one translation of this would result in machine language. The computer considered to be the ultimate receiver of the object code is purely imaginary, as is the symbolic code. However, it is assumed that the machine has index registers, a single accumulator, indirect addressing, and specific commands for the manipulation of characters or bytes. Both binary and decimal internal number representations will be considered.

2

The Formal Definition of Language

A set of rules that defines the formulation of natural language statements is a *grammar,* which unfortunately is not always taught as a complete set of rules in our grade schools, but rather as a disjointed commentary on the shortcomings of a child's expression of thought. Thus the coalescing of a formal grammar is mainly concerned with the task of extracting from the well-known rules, often applied instinctively, sufficient data to present a formalism that will both adequately model our mode of communication and also allow the future generation of yet unspoken but meaningful thoughts.

In music, tonality controls the construction of a harmony. Though the key signature of a piece of music is evident on the manuscript, the listener needs to have the key established by dominant and tonic chords before other modulating or chromatic progressions are introduced. Thus the student of music is taught the construct rules for the development of satisfying pieces. Similarly, the principle of seriality (in a 12 tone scale and in which no note is repeated), of which Stravinsky is the greatest exponent, establishes the rules that are responsible for the production of more modern music. However, since our ears have become attuned to the principle of tonality, seriality is not always as satisfying to some listeners as tonal compositions.

The Modes of Definition

In the process of learning a new computer (or nonnatural) language, the format of a statement or phrase is often presented in terms of a natural language description that defines the permitted components and required features. For example, the definition of a DO statement in FORTRAN may take the form:

$$\mathsf{DO}\ n\ i = m_1, m_2, m_3$$

where n is a statement identifier, i is a simple integer variable, and m_1, m_2 and m_3 are simple integer variables or constants. In this manner, the syntactic prescription for the writing of a legal DO statement may be described depending on the user's knowledge of several other prescriptions. For example, in this case it is assumed that the reader is familiar with the formalisms that describe statement identifiers, integer variables, and integer constants. However, such a description must be adjoined by a semantic description to permit the reader to formulate a statement that will be compiled to a set of object code, which in turn will execute the desired operations.

In particular, a semantic description must indicate that the statement identifier contained in the DO statement refers to a statement not previously defined in the subprogram, and that the assigned values of the variables or constants replacing m_1, m_2 and m_3 must relate to each other in a specific manner. That is,

$$\mathsf{V}(m_2) > \mathsf{V}(m_1) > 0 \qquad \text{and} \qquad \mathsf{V}(m_3) > 0$$

where $\mathsf{V}(x)$ is the value at execution time of the variable or constant that replaces the pseudovariable x.

While such a description is readable and gives sufficient information for the correct formulation of a statement, the exceptions and alternatives must be given in appended descriptions or in a set of alternate prescriptions.

Semantic description of statements has not reached the state where a simple prescription can be given in a manner that is readily understandable to all users. Elgot and Robinson † have proposed a machine model to simulate some of the basic features of the central processing unit of a modern digital computer. Using the notion of a random access-stored program machine (RASP), they consider the relationships between a problem-oriented language and machine language. However, little further work has been published that would allow the semantic definition of language components in a unified, concise manner.

Syntactic definition, contrariwise, has been successfully developed in a complete, efficient and concise form. In fact, the success of the manner of definition can be judged from the diversity of language descriptors that have been developed from a single original proposition. ‡ The formal definition of syntax is more compact and less ambiguous than a similar definition in conventional text material. However, this does not imply that, by definition, a formal definition cannot produce an unambiguous result. While an English language description with many *if . . . then, or* and *and* connectives can be ambiguous in that the reader is left bewildered as to what procedure to follow to formulate a legal statement, the formal definition may clear up this dilemma. But two definitions may allow the formulation of the same sequence of characters. In the same manner, formal definitions of the syntax of a natural language may allow the construction of the same sentence from differing specifications. For example, those English words that perform double duty as both verb and noun, such as *rose, bow* and *list,* or exist as both an adjective and verb, such as *live,* can cause the evolution of an ambiguous creation. However, although ambiguity of definition can be eliminated, the elimination of ambiguity in determining the origin of an object is difficult. For example, taken out of context, the written word *read* can have multiple meanings, and the impingement of the same word on the ear can engender thoughts of a color or a plant. Thus to say that the origin of the word was from the definition of a verb, noun, or adjective is impossible. Similarly, FORTRAN has sequences of digits which, when taken out of context, can either represent an unsigned integer or a statement number.

† C. C. Elgot and A. Robinson, "Random Access-Stored Program Machines, An Approach to Programming Languages," *Jour. A.C.M.,* Vol. 4, Oct. 1964. Since the first edition, significant steps in this direction have been formulated by the Vienna Laboratory of IBM.

‡ C. Backus, "The Syntax and Semantics of the Proposed International Algebraic Language," UNESCO Conf. on Info. Proc., Paris, 1959.

While semantic and pragmatic ambiguities are of considerable interest, they are without the scope of this text.[†] Only semantic ambiguity will be our concern, and then only with respect to its avoidance.

Language Descriptors

The language in which a language may be defined is termed a *metalanguage* and must be uniquely distinguishable from the language being described. Thus attempts to define a language in terms of itself can lead to paradoxes due to the indistinguishability of the metalanguage and the language. For example, we may say in the metalanguage of English that a sentence has certain qualities, such as *it is grammatically correct* or *that sentence is true*. Consider then the sentence: *This statement is false.*[‡] If, not being given the information as to whether this sentence is written in the language or metalanguage, one assumes that the word *this* refers to the statement itself, then the sentence is paradoxical. However, the same utterance on the part of a scholar pointing to some other statement is clearly valid. Thus the metalanguage for ALGOL, for instance, must be clearly distinguishable from ALGOL. By these requirements, the symbolism of a metalanguage must not include the symbols used in ALGOL.

To formalize the definitions in the metalanguage, each definition is given the form of a statement or *construct,* which is analogous to a formula. However, to accomplish some unique features of such a specification, the operators define a mode of construction, or *concatenation.*[§] In this text we shall employ the following symbols in the metalanguage:

$<x>$	the object named *x*
:=	. . . can be formed from . . .
|	or (the exclusive or)
$\{z\}_i^j$	*z* is to be repeated at least *i* times but not more than *j* times. When *i* is omitted, its value is to be assumed to be 1, and when *j* is absent, its value is assumed to be infinity.
[. . .]	a reducing set (see a later description)

[†] See P. L. Garvin, Editor, *Natural Language and the Computer,* McGraw-Hill Book Co., New York, 1963.

[‡] B. A. R. Russell, *Principia Mathematica,* 3 Vols., 1910–1913.

[§] From *concatenate, v.t., to join or link together; connect in a series.* Standard College Dictionary, Funk & Wagnall, 1966.

Further, for the sake of clarity, another convention of typesetting will be adopted. Those characters which are to form part of the language being described will be set in sans serif type, such as A, B, C . . . , while the names of objects (enclosed in corner braces <>) will be set in italics. Thus an object appearing outside the corner braces in sans serif type is explicitly a language character. The corner braces used to parenthesize a component name will not become confused with the *less than* or *greater than* symbols: The metalanguage symbols occur in pairs with their open ends facing each other, whereas the *less than* and *greater than* symbols are obviously disjointed. For example, there is no confusion between the string >|<, which is taken to mean *the symbol for greater than* or *the symbol for less than,* and the string <|>, which defines a metalanguage name |.

The format of a metalanguage construct will be (in the *meta*-metalanguage of English) as follows:

> *The object named in the corner braces may be formed from the objects named or specified on the right.*

This definition specifically avoids any reference to concatenation on the right-hand side of the construct, since not all constructs contain the operation of concatenation, and where desired, the concatenation operator is specified. In fact, concatenation is implied by the juxtaposition of names or objects in the construct. Thus the metalanguage construct

A<*x*>;

is intended to symbolize the linear concatenation of the object A, the object named *x* and a semicolon. If <*x*> had previously been defined as any single digit, then a legal construct of A<*x*>; would be

A1;   or   A9

but not   A*x*;   or even   AX

since neither *x* nor X is a legal replacement of <*x*>.

To signify alternatives in the construct, the *or* symbol is used. Thus a decimal digit might be defined by the metalanguage statement:

<*decimal digit*> := 0|1|2|3|4|5|6|7|8|9

which is taken to mean

> *the object named "decimal digit" may be formed from any one of the characters* 0, 1, 2, 3, 4, 5, 6, 7, 8 *or* 9.

Note that while the names used in corner braces are generally chosen in this text to be indicative of the semantic nature of the resulting accretion, in fact, they are merely a collection of marks that are distinguishable one from another. Thus the above definition might well have been written:

<*dd*> := 0|1|2|3|4|5|6|7|8|9

Two definitions cause problems in the metalanguage, and thus will be explained in the meta-metalanguage:

<*null*> :=
<*blank*> := b

The first definition above declares that the object named *null* is to be formed from a nonexistent character. That is, the item is absent from the language being described. This gives us the opportunity to state in the metalanguage such statements as

. . . *the third index may be omitted.*

In particular, consider the statement from a FORTRAN text:

> *Negative constants are prefixed with a* – *sign; positive constants may have a* + *sign, but do not require it.*

If an unsigned constant has been predefined and given the name <*constant*>, then a signed constant may be defined by the construct:

<*signed constant*> := <*sign*><*constant*>

where a <*sign*> has been defined by the construct

<*sign*> := <*null*>|+|–

which indicates that a <*sign*> may be chosen from either of the symbols + or –, or may be omitted.

The definition of <*blank*> is necessary when it is desired to make the metalanguage both readable and unambiguous. For example, take the above definition of <*sign*>. Although this definition appears in print as a number of characters followed by a white space to the right-hand edge of the page, it is in fact typeset using a special set of slugs that leave no mark on the paper. This lack of marks leaves one in a quandary as to whether the blank belongs to the set of italicized characters or the sans serif set. So, in the definition, we are uncertain as to whether a <*sign*> is defined as the <*null*> object, a + sign or a – sign followed by a large number of blanks.

To overcome this ambiguity, we shall specify that printed blanks are not part of the metalanguage and will be ignored, except when such a blank is a mandatory part of the language being described. When a blank is to be included in the accretion, the mark b or the name <*blank*> will be used. Thus in the language being described, the object produced will be " ", while in the metalanguage the mark denoting the same mark will be b, which is given the name <*blank*>. For a similar distinction between the object, the symbol for the object, and the name of the object, see the conversation between Alice and the White Knight in Lewis Carroll's *Alice through the Looking Glass.*

Recursion and Repetition

The definition of some portions of a language give so many options that the specific definition of the statement by the concatenation of only those items listed in the construct becomes unwieldy, if not impossible. For example, many statements in computer languages contain a list of unspecified length, each item of which is a variable, the items being separated by commas. To define such a list as

<*list extender*> := ,<*variable*>|<*null*>

<*list*> := <*variable*><*list extender*><*list extender*><*list extender*>

not only prescribes a limit to the number of items in the list, but also fails to be concise. A technique of recursive definition solves this problem both concisely and without recourse to special symbolism. Thus the above two definitions may be consolidated to:

<*list*> := <*variable*>|<*list*>,<*variable*>

Such a construct would seem to be redundant, for it contains two alternatives, only one of which can be used since, at the outset of using the construct, a <*list*> does not exist. However, if the first of these alternatives is regarded as a starter and the other as an expander, then one can use the <*list*> chosen from the starter to form other longer <*list*>s by using the expander. Some linguists have objected to the use of a recursive definition and, as an alternative, have introduced a set of symbols indicating the repetitive concatenation of a given set of objects. Such repetitive concatenation is not restricted to the addition of objects to either the right-hand or left-hand end of an existing accretion. However, the repetition is to be performed in situ so that the substring is formulated before the remainder of the statement (if any) is concatenated. The symbolism for repetition is $\{z\}_i^j$ where the subscript and superscript define the limits on the number of repetitions to be performed. Examples of the use of repetitive braces include:

$\{A\}_1^3$	can be expanded to	A,AA, or AAA
$\{AB\}_1^2$		AB or ABAB
$\{A\}$		A,AA,AAA, . . . or a string of As of any length. The metastatement is equivalent to $\{A\}_1^\infty$.
$\{AA\|A\}_1^3$		A,AA,AAA,AAAA,AAAAA or AAAAAA
$\{A\|B\}_1^2$		A,B,AB,BA,AA or BB
$X\{A\}_1^3Y$		XAY,XAAY or XAAAY
$X\{A\}_0^2$		X,XA or XAA
$\{A\|B\|C\|D\}^1$		A,B,C or D The metastatement is equivalent to $\{A\|B\|C\|D\}_1^1$

With this mode of specification, a <*list*> may be defined as:

$$<list> := <variable>\{,<variable>\}_0^\infty$$

This type of definition has the advantage that the size of an accretion may be defined explicitly, if so desired, while a conciseness that is not possible without the use of a recursive definition is maintained. As an adjunct to recursion, but without adding a complete new metalanguage definition, Ledley [†] has suggested the use of a qualified :=. He has used the notation of a superscripted :=, in which the superscript defines the maximum number of times that the construct may recur to formulate the object. Thus

$$<list> \overset{6}{:=} <variable>|<list>,<variable>$$

may construct an accretion in which the object named <*variable*> occurs no more than six times, but at least once.

The use of repetitive braces with the limits 1 and 1, though not essential, allows the reduction of the number of metastatements in a formal statement

[†] R. S. Ledley, FORTRAN IV Programming, McGraw-Hill Book Company, New York, 1965.

with several alternatives at a secondary level. For example, in IBM 1620 GOTRAN,† the maximum arithmetic assignment statement consisted of a simple variable on the left-hand side and an expression on the right containing no more than one arithmetic operator. If <*variable*> and <*number*> have been predefined, and one does not use the repetitive brace notation, an assignment statement in GOTRAN would have to be defined by the metastatements:

$$\begin{aligned}&\langle \textit{binary operator}\rangle := +|-|/|*|**\\ &\langle \textit{unary operator}\rangle := +|-|\langle \textit{null}\rangle\\ &\langle \textit{element}\rangle := \langle \textit{variable}\rangle|\langle \textit{number}\rangle\\ &\langle \textit{assignment statement}\rangle := \langle \textit{element}\rangle\langle \textit{binary operator}\rangle\langle \textit{element}\rangle|\\ &\qquad\qquad\qquad\qquad\langle \textit{unary operator}\rangle\langle \textit{element}\rangle\end{aligned}$$

With the repetitive brace notation, the same set of definitions may be condensed to a single metastatement:

$$\begin{aligned}\langle \textit{assignment statement}\rangle := \{&\langle \text{variable}\rangle|\\ &\langle \textit{number}\rangle\}^1\{+|-|/|*|**\}^1\{\langle \textit{variable}\rangle|\\ &\langle \textit{number}\rangle\}^1|\{+|-|\langle \textit{null}\rangle\}^1\{\langle \textit{variable}\rangle|\\ &\langle \textit{number}\rangle\}^1\end{aligned}$$

Both the notations of repetitive braces and restricted recursion contain the disadvantage of not being linear definitions and therefore are not immediately suitable as input to a syntax-oriented translator.

The indices of repetitive braces can be variables or expressions, provided the value of the variables is defined within the construct. For example, although a statement number ‡ (in FORTRAN) can be defined as:

$$\langle \textit{statement number}\rangle := \{\langle \textit{digit}\rangle\}_1^5$$

where $\langle \textit{digit}\rangle := 0|1|2|3|4|5|6|7|8|9$

this definition is only applicable to the statement number that occurs within the body of a statement. Thus if we distinguish between a statement number and a statement identifier (in this discussion) by stating that a statement number occurs only in a statement body whereas a statement identifier

† Reference Manual, IBM 1620 GOTRAN. Interpretive Programming System, Form No. C26-5594-0, IBM, 1961.

‡ This particular definition will permit the construction of zero statement numbers, which is not permitted, whereas leading zeros are permitted provided at least one digit is nonzero. The definition of a nonzero statement number is left as an exercise for the reader.

precedes the statement body, then a FORTRAN statement may be defined as being in the form:

$$<\mathit{FORTRAN\ statement}> := <\mathit{statement\ identifier}>\mathsf{b}<\mathit{statement\ body}>$$

where a $<$*statement identifier*$>$ is to be exactly five characters in length, to conform with the card format required on input to the compiler. If we insist that the digits within a statement identifier be right justified in the field of five characters, the following definition will suffice:

$$<\mathit{statement\ identifier}> := \{<\mathit{blank}>\}\,_{5-n}^{5-n}\,\{<\mathit{digit}>\}\,_{0}^{n\leqq 5}$$

whereas, if the number may occur anywhere in the field:

$$<\mathit{statement\ identifier}> :=$$
$$\{<\mathit{blank}>\}\,_{5-m-n}^{5-m-n}\,\{<\mathit{digit}>\}\,_{0}^{n\leqq 5}\,\{<\mathit{blank}>\}\,_{0}^{m\leqq 5-n}$$

Using this metalanguage, let us now define a simple language. As an example, consider the language of propositional logic in Polish String notation [†] as defined by Allen [‡] :

A given expression is a WFF (*W*ell *F*ormed *F*ormula) if and only if:

(a) it is a 'p', 'q', 'r', 's'; or

(b) it is a two unit expression in which the first unit is 'N' and the second unit a WFF; or

(c) it is a three unit expression in which the first unit is a 'C', 'A', 'K', or 'E', and the second and third units are WFF's.

These definitions may be consolidated to the construct:

$$<\mathit{WFF}> := \mathsf{p}|\mathsf{q}|\mathsf{r}|\mathsf{s}|\mathsf{N}<\mathit{WFF}>|\{\mathsf{C}|\mathsf{A}|\mathsf{K}|\mathsf{E}\}^{1}<\mathit{WFF}><\mathit{WFF}>$$

Since a language must be constructed from the building blocks of the characters, the most common components of the statements must be defined sequentially early in the total definition, so that a program may be defined on this basis. In fact, since each definition in the metalanguage is to be used to direct the syntactical writing of a program, the metalanguage must eventually define a program. Thus, in examining the interaction of metalanguage statements with one another, there must exist a series of links starting with the exposition of the character set of the language (which may in fact appear in more than one statement) and culminating in a

[†] See Chapter 7 for a detailed description of Polish String Notation.
[‡] L. E. Allen, *WFF n'PROOF*, Yale Law School, 1962.

single definition. All links must be continuous so that there exists only one objective after concatenation.

Reducing Sets

A type of definition that is difficult to convert into the metalanguage is of the form:

> *A widget may be formed by the concatenation of one to six different digits.*

Now since a *widget* is constructed from digits, the choice of characters is restricted to one each of the set:

$$0,1,2,3,4,5,6,7,8,9$$

That is, once a character has been used, it is no longer available as a component of a *widget.* To permit such a choice, let us define a *reducing set,* symbolized by square brackets, in which it is understood that once an element has been used, the set is reduced by that element. That is, in terms of set theory,

$$\begin{aligned}&\text{if } \mathsf{N} = \{0,1,2,3,4,5,6,7,8,9\}\\ &\text{and } a \in \mathsf{N}\\ &\text{then } \mathsf{R} = {\sim}(\mathsf{N} \cap a)\end{aligned}$$

where N is the original set, a the chosen element and R the reduced set. However, the reduced set becomes the set from which the next choice may be made. Thus the choice of characters to construct a *widget* may be described by the following algorithm:

if

$$\mathsf{N} = \{0,1,2,3,4,5,6,7,8,9\}$$

then

(choice 1) $a \in \mathsf{N}$ and $\mathsf{R1} = {\sim}(\mathsf{N} \cap a)$
(choice 2) $b \in \mathsf{R1}$ and $\mathsf{R2} = {\sim}(\mathsf{R1} \cap b)$
(choice 3) $c \in \mathsf{R2}$ and $\mathsf{R3} = {\sim}(\mathsf{R2} \cap c)$
(choice 4) $d \in \mathsf{R3}$ and $\mathsf{R4} = {\sim}(\mathsf{R3} \cap d)$

and so forth.

Using this notation, one sees that a *widget* may be defined by the construct:

$$<\textit{widget}> := \{[0|1|2|3|4|5|6|7|8|9]\}_1^6$$

In use, this metadefinition operates as follows:

Step	*Choice*	*Widget*	*Remaining Set*
0			0,1,2,3,4,5,6,7,8,9
1	3	3	0,1,2,4,5,6,7,8,9
2	7	37	0,1,2,4,5,6,8,9
3	0	370	1,2,4,5,6,8,9
4	1	3701	2,4,5,6,8,9
5	9	37019	2,4,5,6,8

This metalanguage definition will permit, for example, the definition of PL/I DECLARE statements, wherein the attributes of a name may occur in any order, but may not be repeated. In particular, consider the case where in defining the file attributes, the programmer is permitted to define a file option and/or a key option in any order, neither of which elements are mandatory. The file attribute may be defined as:

$$<\textit{file attribute}> := \{[<\textit{file option}>|<\textit{key option}>]\}_{0}^{2}$$

Context Dependency

In the use of syntax constructs, the progression from the initial metavariable to the actual string of characters may be visualized as the progressive substitution of metavariables by their components (which may contain further metavariables) until all metaelements have been replaced by elements of the character set of the language. This may be further visualized as the progression through certain branches of a tree structure wherein each branch is independent of all other branches. However, this tree-like structure with no interdependence of branches only exists for *context free* languages. If the left-hand side of a construct contains more than one metavariable, then the production of the right-hand side is dependent on the occurrence of more than one metavariable, and the language is said to be *context sensitive.* In such languages, constructs of the type:

$$<a><b> := <a><c>$$

indicate that when the metavariables $<a>$ and $<b>$ occur *in that order,* then the metacomponent $<b>$ may be replaced by the metavariable $<c>$. For a discussion and formal definition of context-sensitive languages, see Ginsburg.†

† S. Ginsburg, *The Mathematical Theory of Context Free Languages,* McGraw-Hill Book Co., New York, 1966.

Summary

In this chapter we have described a metalanguage that will enable us to define nonnatural languages, such as those used in both computing and logic. However, we have only discussed the definitions with respect to the formulation of a statement of the object language and not with regard to the determination of the name of a string of characters with which we are presented. That is, we have been given the key to the creation of a cipher, but have not checked to see whether it is the same key that will allow us to decipher the message.

Problems

2.1 Write definitions that will permit the construction of odd and even integers.

2.2 In a FORTRAN program, the format of the input data is described by the following statement:

```
FORMAT(F16.3,I7,4I6,3X,6HOUTPUT)
```

Assuming that the read statement referring to this FORMAT contains the names of six variables of the appropriate mode, write a set of constructs that will describe the valid forms of preparing the input data documents.

2.3 A standard subscript in a FORTRAN array variable cannot exceed the metaexpression c*v±k, where c and k are integer constants and v is an integer variable. Write a construct describing subscript expressions which contains a minimum number of alternatives. That is, condense to a construct containing a minimum number of occurrences of the metalanguage operator "or," the construct:
$<sub\ exp> :=$

$$<c>*<v>\{+|-\}^1<c>|<c>*<v>|<v>\{+|-\}^1<c>|<v>|<c>$$

2.4 Write a construct that will describe a nonzero integer number.

2.5 Write a construct that will describe a sterling constant and consists of the following concatenated fields:

(a) a pounds field that is a decimal integer,

(b) an oblique stroke (/),

(c) a shillings field that is a decimal integer less than 20 with no leading zero or blank, and in which a zero field is written as —,

(d) an oblique stroke,

(e) a pence field that is a decimal integer less than 12 with no leading zero or blank, and in which a zero field is written as –,

(f) L.

2.6 Find the algorithms that enable the prediction of the next letter in each of the following sequences, and then describe these algorithms in terms of the metalanguage such that infinite sequences can be generated from cyclic Roman alphabets (i.e., A follows Z after each pass through the alphabet):

(a) ABABAB . . .

(b) ATBATAATBAT . . .

(c) DEFGEFGHFGHI . . .

3

The Reason for a Grammar

The purpose of expressing either a natural or nonnatural language in a specific form is not primarily centered around our innate desire to conform, but rather on the need to communicate. However, the success of communication is measured by the amount of information that is gleaned by the receiver. Thus, although a phrase may be grammatically correct (such as *convict an indulgent mandrake* or *a nice derangement of epitaphs*†), the listener or reader may gain little information, and therefore, for the sake of conversation, reply in like manner.

The use of grammatically correct sentences is therefore merely a prelude to the deconcatenation of the statement by the receiver. A grammar or formalism that allows the generation of a statement which is not deconcatenable or is deconcatenable in more than one way is syntactically am-

† Mrs. Malaprop, *The Rivals,* R. B. Sheridan.

biguous. In natural language syntactic ambiguity is not always disastrous since the meaning of a certain word may be derived from the context or (as in a malapropism) from the knowledge of a similar word that would make the phrase meaningful. Similarly, permissible errors that allow a phrase to be formed without the loss of meaning are nondisastrous. As an example, read any cablegram couched in terms that minimize the number of words (and hence the cost of the message) while maintaining the meaning of the message. Ambiguity resulting from the inability to deconcatenate a phrase due to a lack in the definition of the scope of an adjective in a natural language is similarly important, though many such ambiguities are resolved by a knowledge of normal (as opposed to standard) usage.

Parsing

The deconcatenation or *parsing* of an accretion to determine the grammatical correctness of the statement is not only dependent on the individual rules of the grammar or metalanguage, but also on the interrelations of those rules.

The result of using a construct (that is, a set of syntax rules) is merely a collection of symbols or marks which, through practice, can be deconcatenated to allow one to extract the meaning (if not the intent) of the message. If the same message were formulated from the same constructs but with a different set of characters, then the message would be without meaning since it could not be parsed by sight. On the other hand, an expert decoder, such as a telegraph operator, can parse a message in a nonnatural character set, such as morse code. Similarly, computer operators who work continually with paper tape become adept at reading the holes in the tape.

As a language formalism must start with a definition of the character set from which that language is built, so the process of parsing must commence with that which exists and deconstruct to determine the type of statement or, if the type is known, to determine the legality of the character string.

Let us consider the definition of a Well-Formed Formula (WFF) given in Chapter 2 [†]:

$$<WFF> := \mathsf{p}|\mathsf{q}|\mathsf{r}|\mathsf{s}|\mathsf{N}<WFF>|\{\mathsf{C}|\mathsf{A}|\mathsf{K}|\mathsf{E}\}^1<WFF><WFF>$$

[†] In this chapter we shall allow the meaning "... is defined by ..." to be assigned to the symbol := .

and instead of attempting to construct a legal WFF from this construct, let us check the validity of a character string that is assumed to represent a WFF:

EAsKNpNrNq

Now a primary examination of this string shows that there exist four <WFF>'s, which are immediately recognizable, that is, p, q, r, and s. Let us replace these elements of the string by the component name <WFF>:

EA<WFF>KN<WFF>N<WFF>N<WFF>

Reading the string from left to right, we may note that the first character is E, which, indeed, does appear in the construct within the **alternative** †

E<WFF><WFF>

However, the next two elements in the string do not conform to this pattern, so a <WFF> utilizing the character E cannot be constructed at this point. Using similar logic, one can see that the only legal <WFF>'s in the string are those that involve the character N. If these are replaced by the component name as before, the string is reduced to

EA<WFF>K<WFF><WFF><WFF>

Neither the E nor the A can be used to form a <WFF>, but one is recognized at K, that is, K<WFF><WFF>, which after replacement reduces the string to

EA<WFF><WFF><WFF>

The next two steps are obvious:

<WFF> := A<WFF><WFF>

so the string is reduced to E<WFF><WFF>, and since this conforms to the definition of a <WFF>, the string is reduced to the single item

<WFF>

Thus we may conclude that the string EAsKNpNrNq is a legally constructed <WFF>. This process is pictured graphically in Fig. 3.1.

Let us now consider the string CpAqrKs. The graph of the parsing of this string is shown in Fig. 3.2. However, the string is not reduced to a single item, there being left

<WFF>K<WFF>

which does not appear as an alternate construct in the definition of a <WFF>. Hence we must conclude that CpAqrKs is not a legal <WFF>.

† The type of parsing, starting from the string and attempting to construct the syntactic tree back to the root component, is known as "bottom up" analysis.

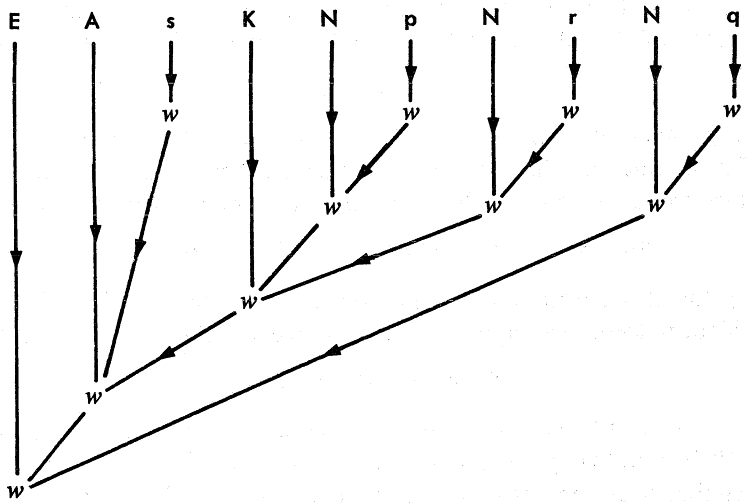


FIGURE 3.1

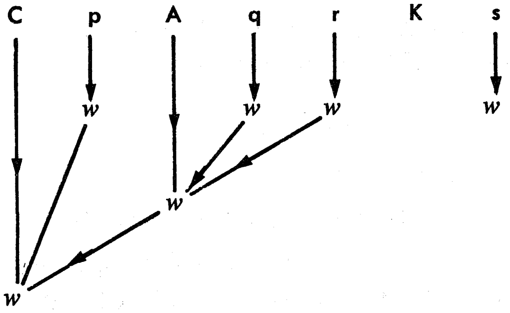


FIGURE 3.2

Problems

3.1 Assuming the definition of a <*WFF*> and using the graphical mode of parsing, check the validity of the following strings, which are intended to be Well-Formed Formulas:

(a) AKrpNq	(d) NNAErpp
(b) CCsCNssr	(e) NArKqEr
(c) ANpCpi	(f) CANAprss

3.2 Given

$$<x> := \mathsf{A}|\mathsf{B}|\mathsf{C}$$
$$<y> := <x>\{0|1|2|3|4|5|6|7|8|9\}^1|<y><y>$$
$$<z> := \{\mathsf{A}|\mathsf{B}|\mathsf{C}\}^1_0 <y>$$

determine the type (that is, $<x>$, $<y>$ or $<z>$ of the following strings:

(a) CC2C3C4
(b) AAB1
(c) B5C7A0
(d) AA2B

Structure within Syntax

Although it is customary to define a language in terms of many constructs, this is really a matter of convenience. Given a large enough piece of paper, a program could be defined in terms of the elemental characters of the language, instead of breaking down the statement to many groups and then rejoining those groups into a single definition. However, where such groupings have qualities that are fundamental to the semantic nature of the language, such a breakdown is both convenient to the formulation of meaningful statements and to the compilation of those statements. Thus, although the complete parsing of a string of symbols can only determine the type of statement for which that string stands, the by-products of the parse can be put to good use by the compiler generators.

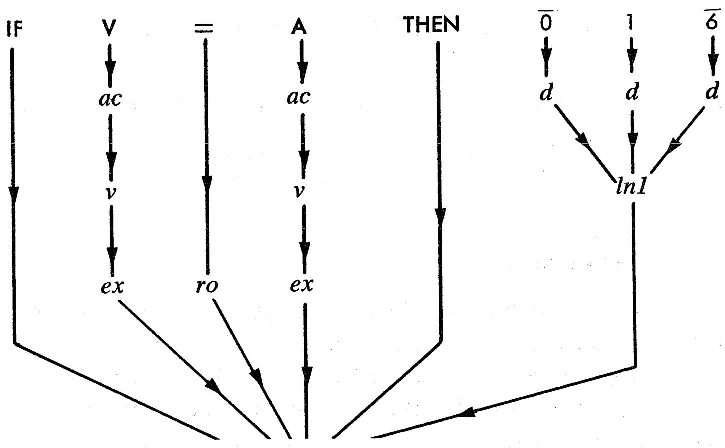
Since a statement to be parsed is presented as a string of characters, the analysis to determine the type of statement must attempt to reconstruct the links between the characters, the components of the language (such as variable names, etc.) and the statement. When the type of a statement has been determined, or a statement can be of only one type, the reconstruction of the linkages is comparatively straightforward, even when the accretion is invalid. However, when a program is defined as a set of statements with many alternate choices, an invalid statement of one type may be a valid statement of another type.

Consider the formal definition of Dartmouth BASIC, given in Appendix C. A BASIC IF statement is defined by the construct

$$<\textit{IF statement}> := \mathsf{IF}<\textit{expression}><\textit{relation op}>$$
$$<\textit{expression}>\mathsf{THEN}<\textit{line no } 1>$$

where the simplest $<$*expression*$>$ is defined as a $<$*variable*$>$, whose simplest form is an $<$*alphabetic character*$>$, where a typical $<$*relation op*$>$ is the symbol =, and where a $<$*line no* 1$>$ may be a three-digit number.

The parsing of an IF statement would be of the form:



<IF statement>

where *ac* represents the meta component *<alphabetic character>*; *v* is *<variable>*; *d* is *<digit>*; *ex* is *<expression>*; *ro* is *<relation op>*; and *lnl* is *<line no 1>*.

Now if the intermediate conclusions of the parse are discarded as each component is incorporated into a new component, then the one result of the parse will be the conclusion that the string is a legal form of an *<IF statement>*. Providing the intermediate conclusions of the parsing algorithm to the appropriate generator, can save the reanalysis of the string to extract the data necessary for generating object code instructions and reserving memory space for the variables. Also, since the *<IF statement>* is composed of portions of other types of statement, it is likely that the generator is not self-contained, but will call upon other generators to compile the statement components. In particular, since *<expression>*'s are to be found in many other statements, the generator for *<expression>* will not be the sole property of the arithmetic statement generator, but will be a utility routine available to all generators. Similarly, the IF generator can link to the GO TO generator to compile the phrase THEN<*line no* 1>.

Consider an arithmetic expression composed only of simple variables. Any pair of variables may be connected with any one of the operators +,−,*,/ or ↑, or a single operand may be prefixed by a unary operator. However, no two operators may occur in juxtaposition, so that an infix operator may not be followed by a prefix operator, except when the prefix operator and its operand are enclosed in parentheses. Similarly, parentheses may enclose any operand to clarify or to define the hierarchy of calculation. According to these rules, an *<expression>* may be defined by the sequence:

<prefix operator> := +|−|*<null>*
<infix operator> := +|−|/|*|↑
<term> := *<variable>*|*<term><infix operator><term>*|
(*<expression>*)
<expression> := *<prefix operator><term>*

This sequence of statements permits the construction of only legal *<expression>*'s and, in particular, does not permit the concatenation of two operators without the presence of enclosing parentheses. Consider the string A + (−B) * C, for which two parsings are shown in Figs. 3.3 and 3.4. Both parsings conclude that the string is a legal *<expression>*, but one of them does not represent the correct ordering of the operations. Thus if the parsing is used to indicate the sequence of arithmetic operations, then

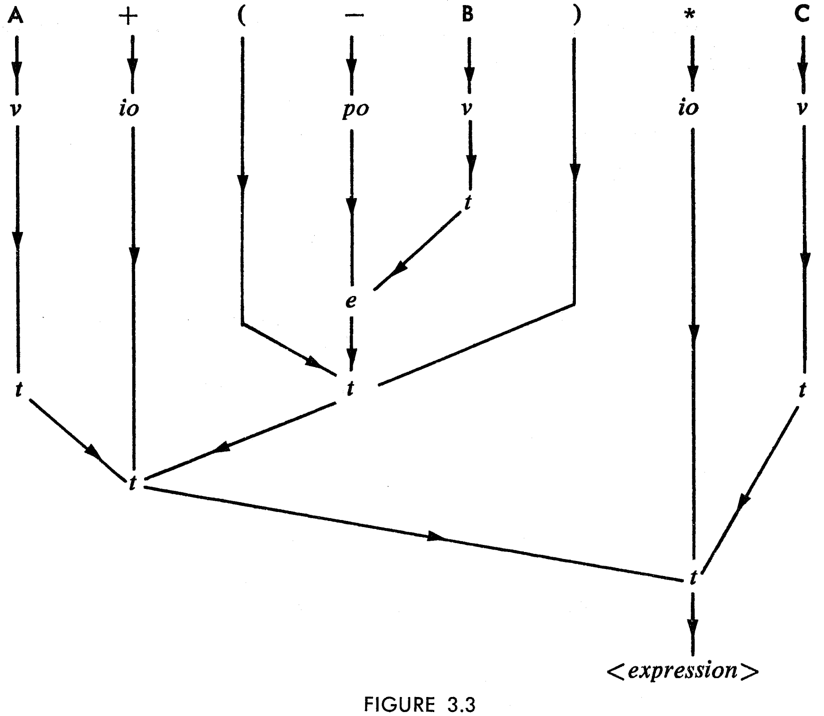


FIGURE 3.3

as terms connected by operators are recognized so that the sequence <*term*><*infix operator*><*term*> forms another <*term*>, the internal structure of the new <*term*> is of no importance in the remainder of the parsing. If a single value replaces the internal structure of that <*term*>, the remainder of the parsing is unaffected. Thus once the string has been determined to be a legal <*expression*>, a trace of the parsing will allow the compilation of each <*term*> as it is recognized. Since the result of a computation in the computer is placed in the accumulator (signified herein as ACC), the first parsing of A + (−B) * C may be broken down to the operations:

Operations	
Generated	*String*
	A + (−B) * C
B → ACC	A + (−ACC) * C
−ACC → ACC	A + ACC * C
A + ACC → ACC	ACC * C
ACC * C → ACC	ACC

Disregarding the original string, let us now reconstruct the string from these operations, parenthesizing each result:

	Generated
Operations	*String*
B → ACC	(B)
−ACC → ACC	(−(B))
A + ACC → ACC	(A + (−(B)))
ACC * C → ACC	((A + (−(B))) * C)

Reducing the number of parentheses to a minimum for readability, we have the string (A + (−B)) * C, which is not equivalent to the original string because of the hierarchical rules of arithmetic computation.

Consider the second parsing of the string and execute the same processes as before:

Operations	
Generated	*String*
	A + (−B) * C
B → ACC	A + (−ACC) * C
−ACC → ACC	A + ACC * C
ACC * C → ACC	A + ACC
A + ACC → ACC	ACC

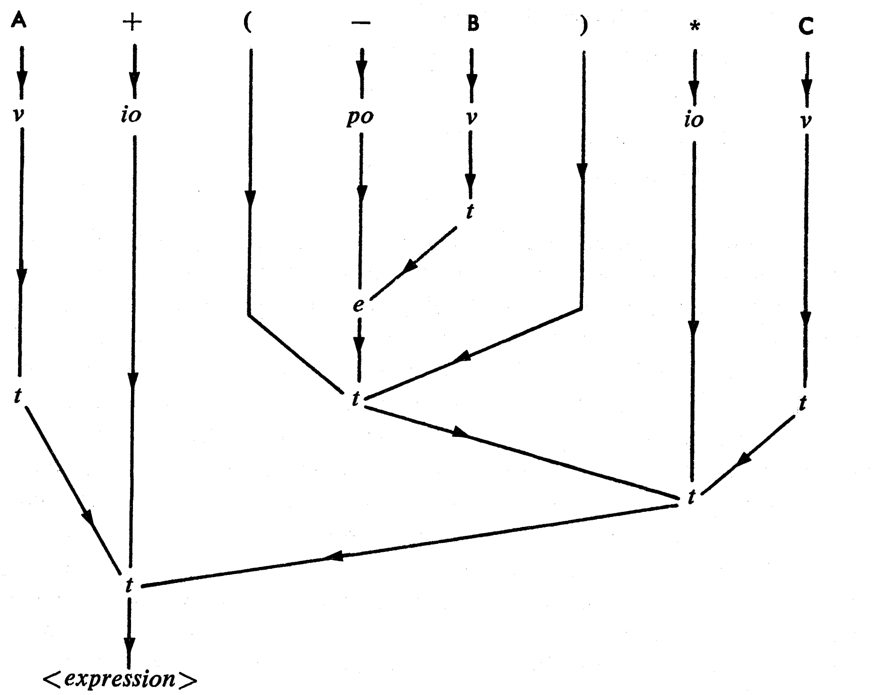


FIGURE 3.4

Operations	*Generated String*
B → ACC	(B)
−ACC → ACC	(−(B))
ACC * C → ACC	((−(B)) * C)
A + ACC → ACC	(A + ((−(B)) * C))
(Deparenthesize)	A + (−B) * C

This latter result is obviously in accordance with the meaning of the original string, but we have no algorithm that will define the correct ordering of the compilation or the more useful parsing. However, give the same string to a high school student and he will have no problem in determining the correct order of computation, because he has been taught a set of rules that define the order:

1. Evaluate parenthesized expressions first.

2. Perform other operations in the order:
   (a) Unary minus (discard unary plus)
   (b) Involution
   (c) Multiplication or division
   (d) Addition or subtraction

With this ordering, the parsing should reveal groupings such that the order of recognition (and hence the order of compilation) is equivalent to the hierarchy of this set of rules. The development of a syntax that will permit the construction of all valid statements and which, at the same time will enforce the parsing by hierarchy of operators is complicated by the syntactical inconsistency of the reverse sign operation (that is, unary minus). By common practice, the syntax of the unary operator differs according to the location of the operator in the string of characters. Since the concatenation of two operators is not permitted, a unary operator and its operand must be enclosed in parentheses within the body of an expression. However, if the same unary operator and operand occur at the left-hand end of the expression, the parentheses may be omitted without the inference that the whole expression is the operand of the unary operator. Thus, though the following expressions are equivalent, the parenthesizing rules are different:

$$-A + B$$
$$B + (-A)$$

The following set of rules permit the construction of valid arithmetic expressions with some measure of hierarchy but without the correct hierarchical placement of the unary operators:

$$\begin{aligned}
&<\textit{term}> := <\textit{variable}>|(<\textit{expression}>)\\
&<\textit{involution factor}> := <\textit{term}>|<\textit{term}>\uparrow<\textit{term}>\\
&<\textit{multiply factor}> := <\textit{involution factor}>|\\
&\qquad\qquad\qquad\qquad <\textit{multiply factor}>\{*|/\}^1<\textit{involution factor}>\\
&<\textit{expression}> := <\textit{multiply factor}>|\{+|-\}^1<\textit{multiply factor}>|\\
&\qquad\qquad\qquad\qquad <\textit{expression}>\{+|-\}^1<\textit{multiply factor}>
\end{aligned}$$

If one uses this sequence of metalanguage statements as a guide to parsing, the following string may be parsed as shown in Fig. 3.5;

$$-A + B$$

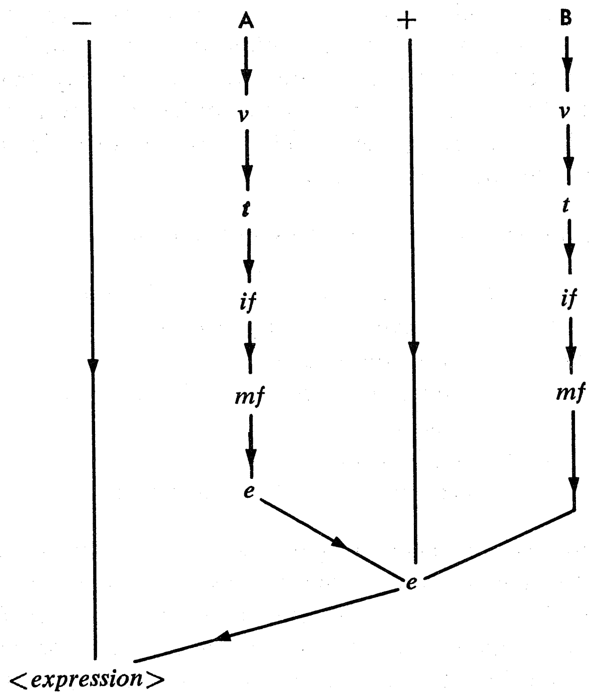


FIGURE 3.5

which compiles to:

Operations Generated	*String*
	−A + B
A → ACC	−ACC + B
ACC + B → ACC	−ACC
−ACC → ACC	ACC

If the string is regenerated from the operations,

Operations	*String Generated*
A → ACC	(A)
ACC + B → ACC	((A) + B)
−ACC → ACC	(−((A) + B))
(Deparenthesize)	−(A + B)

This is obviously not correct; this single deficiency is sufficient to reject this set of syntax rules as a guide to a parsing algorithm. It is left to the reader to verify that parenthesized unary operators are extracted correctly, that is, the expression A + (−B) is parsed in the correct order. However, if an expression preceded by a unary operator will not parse correctly, then neither will a parenthesized unary expression.

Adjusting the metalanguage definition to the set of statements:

(*t*) † <*term*> := <*variable*>|(<*expression*>)|{+|−}^{1}<*term*>
(*if*) <*involution factor*> := <*term*>|<*term*>↑<*term*>
(*mf*) <*multiply factor*> := <*involution factor*>|
<*multiply factor*>{*|/}^{1}<*involution factor*>
(*e*) <*expression*> := <*multiply factor*>|
<*expression*>{+|−}^{1}<*multiply factor*>

This set of metalanguage statements does not satisfy the allowance that a unary operator must be parenthesized within the body of a statement, but may be unparenthesized at the commencement of the statement. In fact, this set allows the concatenation of an infix operator and a unary operator. On the other hand, as guidelines for parsing, this set of metalanguage statements produces a satisfactory breakdown by hierarchy of arithmetic operators. Parsing the same example as before as shown in Fig. 3.6, the compiled instructions are:

Operations Generated	*String*
	−A + B
A → ACC	−ACC + B
−ACC → ACC	ACC + B
ACC + B → ACC	ACC

Operations	*Generated String*
A → ACC	(A)
−ACC → ACC	(−(A))
ACC + B → ACC	((−(A)) + B)
(Deparenthesize)	(−A) + B

† The italicized letters are abbreviations of names to be used later.

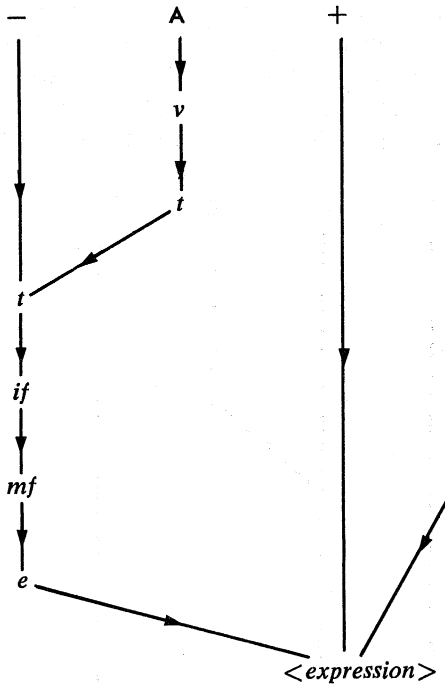


FIGURE 3.6

The Parsing Algorithm

The metalanguage used to define ALGOL, as conceived by Backus, was a unidirectional system that fulfills its original purpose if it permits the construction of only legal statements. However, the extension of the metalanguage to bidirectionality must be approached with trepidation, particularly when the metalanguage statements are to be used as guidelines for both parsing and compiler generator inputs. As shown in the previous example, it is feasible that, under certain circumstances, the construction of a metalanguage statement as a guideline for parsing may not coincide with that for language definition. Moreover, though the metalanguage constructs may be sufficient for the construction of a legal statement, a similar metalanguage statement used for parsing may not be sufficient to define the parsing order.

Consider the example used previously to show the ambiguity of the parsing of the simplest arithmetic expression definition:

A + (−B) * C

By the last definition, the marks +, *, and − must be part of the three constructs: <*expression*>+<*multiply factor*>, <*multiply factor*>*<*involution factor*>, and −<*term*>, respectively. Thus the parsing which determines that the string is a legal <*expression*> is shown in Fig. 3.7. However the parse shown in Fig. 3.8 could have occurred where a link cannot bε

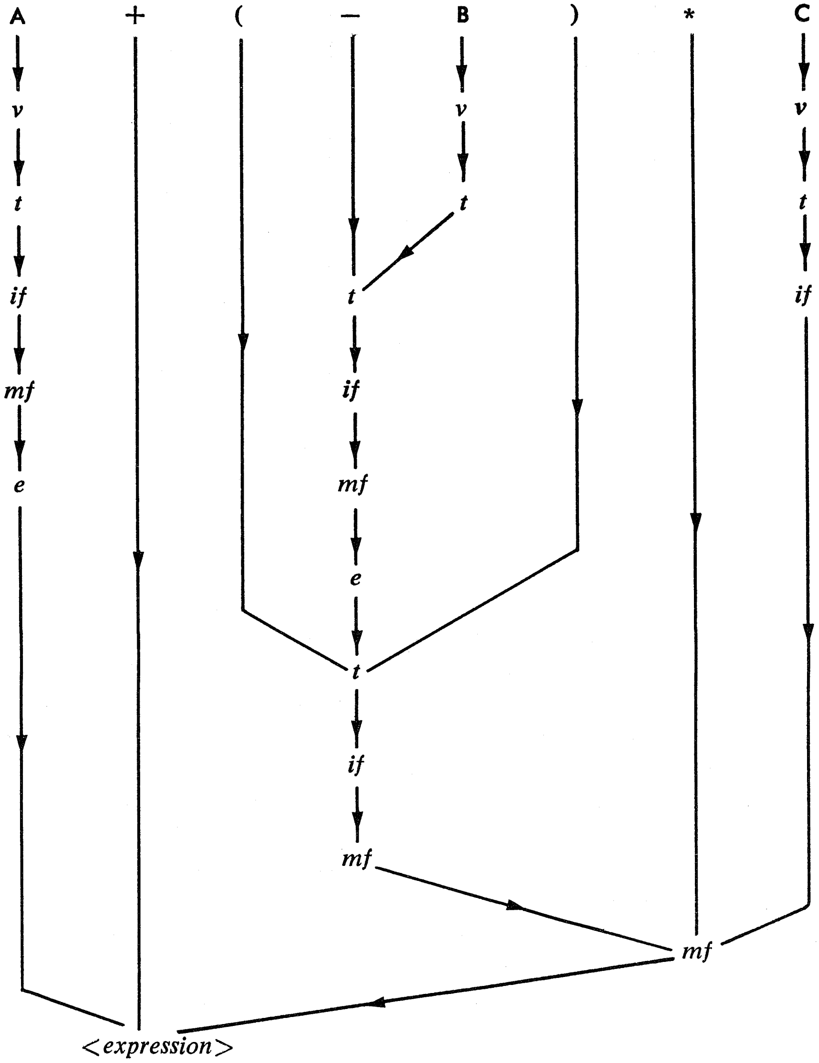


FIGURE 3.7

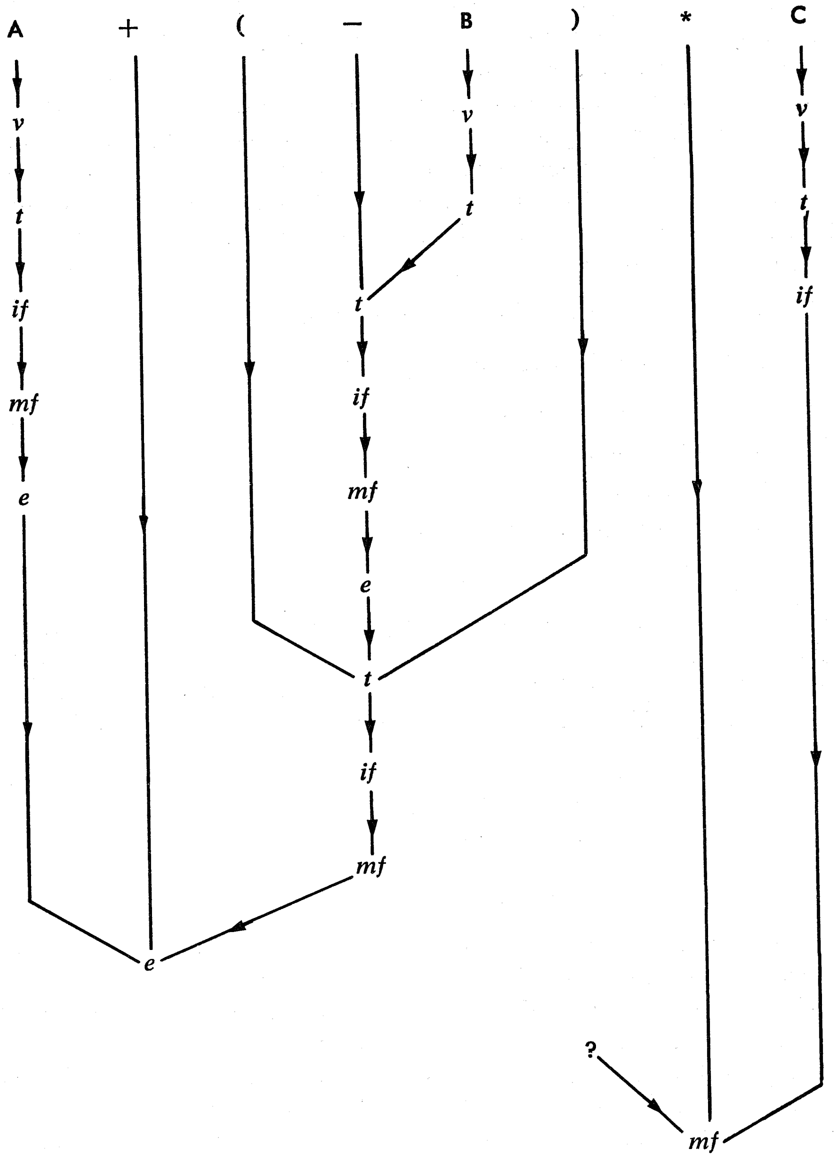


FIGURE 3.8

forged between the <*expression*> to the left and the <*involution factor*> that occurs to the right of the mark *. That is, the components to the lef and right of the * sign are not in accord with the construct. While an <*expression*> can be converted into a <*multiply factor*> by the addition of parentheses, these do not exist in this string in the correct position, and therefore by this parsing one must conclude that the string is not a legal <*expression*>. The obvious difference between these two attempted parsings is the order in which the components of each construct are chosen. Hence, if we can determine and define an ordering for the parse, this ambiguity may be overcome.

The simple rules for evaluating an expression,

1. Evaluate parenthesized expressions first.
2. Perform other operations in the order:
   (a) Unary minus
   (b) Involution
   (c) Multiplication and division
   (d) Addition and subtraction,

were used to determine the sequence of the statements in the metalanguage definition of the <*expression*>; the ordering of the recognition of the components of an <*expression*> must follow the same order. However, in a parse one must add the recognition of a <*variable*>:

<*variable*>
<*term*>
<*involution factor*>
<*multiply factor*>
<*expression*>

Thus if a component highest in the list is formed as soon as its subcomponents are available, then the hierarchical properties of arithmetic statements will be maintained. Further, the intrinsic marks of a component, once recognized, can allow the prediction of the existence of the individual components. For example, in a single scan, a skeletal parsing may be formed for the string A + (−B) * C in which predicted components and subcomponents are determined as shown in Fig. 3.9. In this scan all <*variable*>s have been recognized, and so a second scan must attempt to forge links between the <*term*>'s, the <*variables*>'s, and other marks. Two potential <*term*>'s exist, but it can be seen that one—that is, the one incorporating

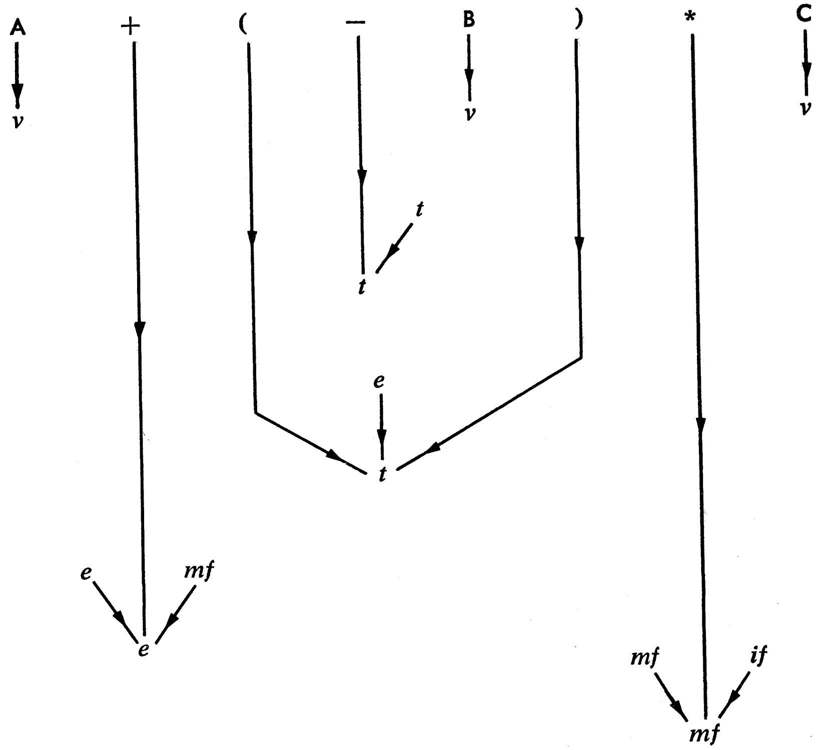
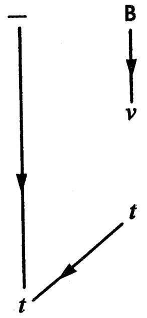
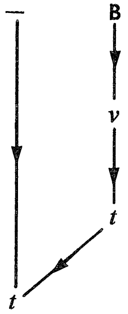


FIGURE 3.9

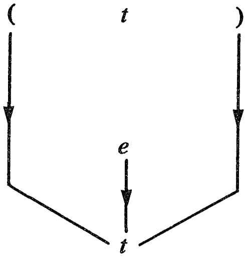
the parentheses and an <*expression*>—envelops the other; thus that of highest stature (with regard to the position on the page) or earliest prediction must be resolved first. Extracting this substring from the string, together with its skeletal parsing:



shows that a link must be forged between the <*variable*> and the <*term*>, which is the component of the predicted resultant <*term*>. From the definition <*term*> := <*variable*> it is obvious that a link can be formed directly between the required component and the existing result.



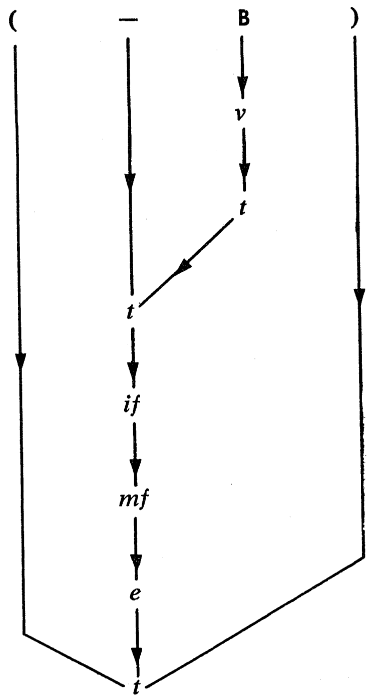
The second level <*term*> may now be determined from the existent string and components:



Following through the set of metalanguage constructs, there is a set of links connecting a <*term*> and an <*expression*>:

<*term*> → <*involution factor*> → <*multiply factor*> → <*expression*>

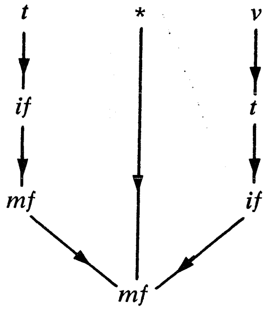
which does not require any additional components. Thus the links in the parsing diagram are



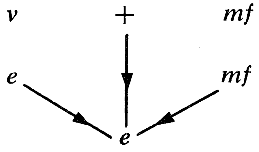
Now, with all <*term*>'s determined for which the components are known, the next level requires the resolution of <*involution factors*>s. None exist due to the absence of ↑ marks, thus the next level will be <*multiply factor*>. The one predicted <*multiply factor*> requires the determination of a left-hand <*multiply factor*> and a right-hand <*involution factor*>, whereas the closest known components on either side are a <*term*> and a <*variable*>. However, links exist that require no other components:

<*term*> → <*involution factor*> → <*multiply factor*>
<*variable*> → <*term*> → <*involution factor*>

Thus the links to the <*multiply factor*> can be determined.



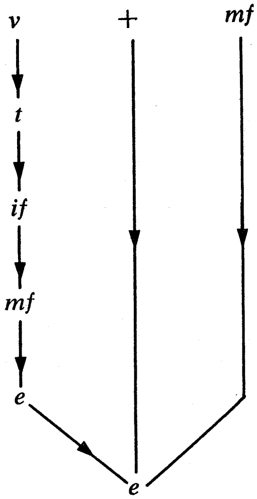
No other <*multiply factor*>s were predicted, and the sole remaining prediction is an <*expression*>, which must be formed from the diagram:



The right-hand component of the <*expression*> must be a <*multiply factor*> and, in fact, such does exist in that position, so that link can be inserted immediately. On the left, a link must be found between a <*variable*> and an <*expression*>, that is,

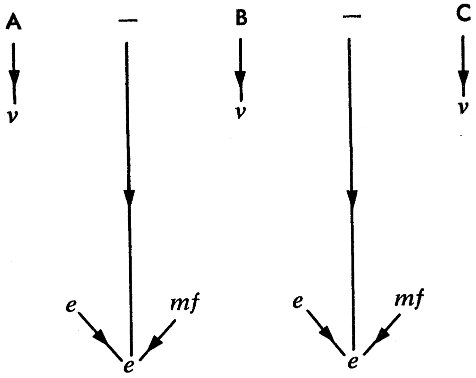
<*variable*>→<*term*>→<*involution factor*>→
<*multiply factor*>→<*expression*>

Hence we have the diagram



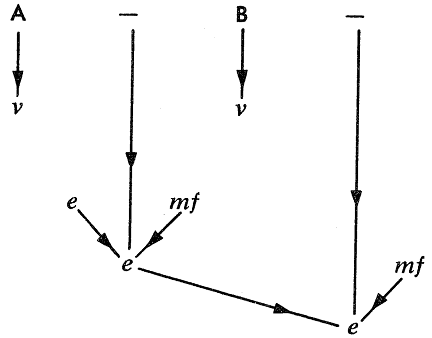
which contains but one result for which there are no further links to any other component.

Where there exists at any one time a set of predictions of equal hierarchical level, the choice of which to resolve first must again be determined by convention that is built into the parsing algorithm. In particular, consider the string



Convention states that a subtraction operation requires that the object to the right of the – sign be subtracted from the object on the left. This fact was built into the metalanguage definition of an <*expression*>, wherein a <*multiply factor*> may be subtracted from an <*expression*>; since a

<*multiply factor*> can be formed from a <*variable*> without the addition of other characters, then individual <*variable*>s can be subtracted from an <*expression*>, but not vice versa. Thus, in terms of the possible links between the metalanguage statements, when two <*expression*>s occur at the same level, they can only be connected by forging a link between either an <*expression*> and a <*multiply factor*>, or a <*multiply factor*> and an <*expression*>. However, an <*expression*> cannot be linked to a <*multiply factor*> without the addition of parentheses, whereas a <*multiply factor*> has a direct link to an <*expression*>, which involves the concatenation of no other components. Thus, in the above example, the <*expression*> predicted by the first − sign (left to right scan), which has a <*multiply factor*> as its right-hand component, can connect with the predicted <*expression*> of the second − sign. Hence the skeletal diagram can be modified to the form



with the assurance that the links shown are the only possible ones. Since the same convention holds for other noncommutative operations, it is true in this particular set of metalanguage statements that the links must be constructed *as soon as* the components are recognized, and that when there is a possibility of an element of the string (or of a resulting component) being linked to two predictions, the link leading to the highest undefined component is to be formed first. Thus in the above example the symbol B could have been linked to either a <*multiply factor*> or an <*expression*>, but by the rule that the link should be forged to the undefined component at the highest level, the <*multiply factor*> is the first choice.

Problem

3.3 The set of syntax rules given on page 48, force the analysis of the string −X↑Y to be parsed effectively as ((−X)↑Y). Amend the syntax analysis rules to force the parsing into the form (−(X↑Y)).

Acceptability Tables

Ingerman [†] advocates the use of *acceptability tables* to determine whether a link can be formed between a given component and a goal. Such a table does not take into account the individual components of a construct, but rather declares that some link can be found. An acceptability table is formed by tagging the rows of a matrix by the names of the components of the metalanguage and the columns by the components and results. The first step is then to indicate in the appropriate element in the matrix where each component is used in the construction of the results. For the metalanguage constructs defining an <*expression*>, given on page 48, the initial table is

	resultants			
	t	*if*	*mf*	*e*
(	1	.	.	.
)	1	.	.	.
↑	.	1	.	.
+	1	.	.	1
—	1	.	.	1
*	.	.	1	.
/	.	.	1	.
v	1	.	.	.
t	1	1	.	.
if	.	.	1	.
mf	.	.	1	.
e	1	.	.	1

where 1 represents that the component is an element of the result (that is, there is a link between the component and the result) and where a period indicates the absence of a direct link. A period is used merely to give some clarity to the table, it being used for the mark 0, which one would expect to find in a binary matrix. This table shows that a component is an essential part of a result, but does not differentiate between the various alternatives. For example, the column under a resultant does *not* indicate that all the components are necessary to form the resultant.

Now since a <*variable*> is a component of a <*term*> (that is, there is a link from <*variable*> to <*term*>), and since <*term*> links with <*in-*

[†] P. Z. Ingerman, *A Syntax Oriented Translator,* Academic Press, New York, 1966.

volution factor> (see row 9, column 2), then there must be a link from <*variable*> to <*involution factor*>. This link must now be entered in the table. To determine all the links in the table, a simple process of performing a Boolean OR operation between various rows of the table can save the effort of chaining through each metalanguage statement. For example, row 3 (↑) shows that there is a link between ↑ and the <*involution factor*>, while row 10 (*if*) shows a link between this component and a <*multiply factor*>. When the result of an OR operation between these two rows replaces the original contents of the highest row (lowest row number) the link between ↑ and <*multiply factor*> is included in row 3. Thus row 3 becomes

	t	*if*	*mf*	*e*
↑	.	1	1	.

The up arrow is now connected to the <*multiply factor*>, which is itself connected to an <*expression*>, which is in turn connected to both itself and a <*term*>. Thus to construct all the links emanating from row 3, it is necessary to operate on row 3 and each other row for which a 1 appears under the appropriate heading. In the particular case of row 3, this eventually entails the operation:

$$\text{row } 3 = \text{row } 3 \;\Lambda\; \text{row } 10 \;\Lambda\; \text{row } 11 \;\Lambda\; \text{row } 12$$

so that row 3 becomes:

	t	*if*	*mf*	*e*
↑	1	1	1	1

In fact, as the reader may verify, the linkage table for the syntax rules of an arithmetic expression contains no zero elements since the cyclic (iterative) nature of the set of constructs allows each component to be linked to every result. This is shown in another way in Fig. 3.10, where the arrows indicate links. Not all acceptability tables are as dense as that for the links between the components and the results of an arithmetic expression; but, in any case, even knowing that it is possible to construct a linkage between an object or component with a result, we cannot determine directly from the table that all components are present in the string to form a result, without reference to the metalanguage statements.

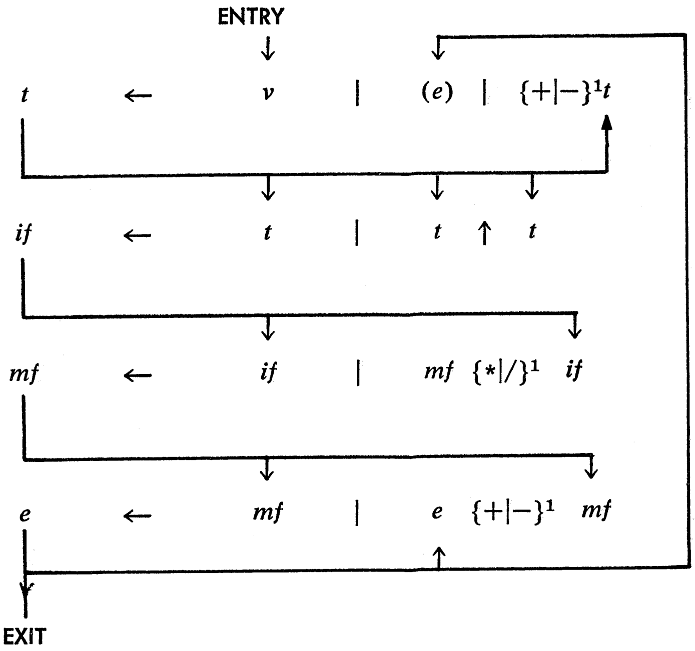


FIGURE 3.10

Problem

3.4 Parse the following strings:

(a) (−(−(−(−(A)+(B*C)))))
(b) A−B*C*(−A*X)/X*(O−R*(O−R*(T−R)))
(c) F−G/((T−M)/S)
(d) −A↑(A/B−C)−D+E
(e) (A−(B−(C−D/E)))

A Specialized Sieve

The determination of statement type and the extraction of language components by parsing, using the guidelines of a syntax definition may be exhaustive and generally applicable, but it is inefficient compared to a specially designed analyzer. In a parse, the whole string of characters must be scanned regardless of the redundancy of this scanning. For example, a BASIC READ statement has the syntax definition:

$$<\mathit{READ\ statement}> := \text{READ}\ <\mathit{variable}>\{,<\mathit{variable}>\}_0^\infty$$

and to determine the validity and type of accretion, the whole string must be examined. However, due to the nature of all BASIC statements, the examination of only the first three characters will enable the type to be determined. The validity of the statement can then be checked as a byproduct of the extraction of the variable names and during the subsequent compilation of the object code.

A close examination of the syntax of a language may reveal that each statement or class of statements has distinguishing marks that set it apart from all others. In other situations, the relationships of the marks within the string to each other determine the type of statement. In particular, the marks that are essential to a statement, such as a comma or parentheses, may be more reliable than variable components that are constructed from other marks and are not essential to the legal construction of a statement.

For example, consider the statements of FORTRAN II. Although FORTRAN is generally taught as having two basic types of statement, that is, declaratives and executables, at compilation time the statements may better be classified as arithmetic and nonarithmetic. It would seem at first glance that the fundamental difference between these types of statements would be the presence of an = sign in arithmetic statements and the absence of that mark in other statements. However, an = sign is a standard occurrence in a DO statement, an indexed I/O statement, and a FORMAT declaration, for example:

```
DO 31 I = J, K, L
READ 100, A, B, (C(I), I = 1, 15, 2)
FORMAT(3HA= , I5)
```

Thus the = sign is not a unique characteristic of the arithmetic statement; but if an = sign is present, the set of possibilities for the identity of that statement has been reduced considerably. By observing the location of the = sign, one may see that in the cases of the READ statement and this particular FORMAT statement, the = sign is contained within a set of facing parentheses, whereas the = sign in both the arithmetic and DO statements is unparenthesized. While the = sign in an indexed I/O statement must have surrounding parentheses, this is not a *requirement* in a FORMAT statement. Consider the statement:

```
FORMAT(5HX)=(Y)
```

Another distinguishing mark in a DO statement, which may be present in arithmetic and FORMAT statements, is the comma. In the DO statement the comma is unsurrounded by parentheses, whereas in an arithmetic statement the comma can only occur within a subscript and is therefore always surrounded by parentheses. FORMAT has no such requirement.

Since FORMAT statements may contain unstructured character strings of any length (according to USASI specifications), and the characteristic marks of the other statements could occur within the FORMAT statement, this statement must be removed from the list of possibilities at the outset of the sieve. This is achieved by using the only known characteristics of the FORMAT statement, that is, the characters F, O, R, M, A, T and (, as the first seven nonblank leading characters. Once this statement has been eliminated, the other statements that may contain an = sign can be distinguished from each other by the following decision table.

	Mark	
	=	,
Arithmetic statement	U	P†
DO statement	U	U
Indexed I/O statement	P	P

† if present
P = parenthesized
U = unparenthesized

The other FORTRAN II statements may be recognized if their keywords, which must always occur in the leading position in the statement, are examined. The list given in Table 3.1 shows the characteristics within the keyword of each statement; the significant characters are upper case, while those that do not materially assist the sieve are lower case. Although the syntax of FORTRAN II insists that the keywords must be present as de-

TABLE 3.1

ACcept
ACcept Tape
ASsign
CAll
COMmon
CONtinue
Dimension
ENd
EQuivalence
EXternal
Function
Go to
Go to (
IF
INteger
INteger FUNCTION †
PAuse
PRInt
PROgram
PUnch
PUnch Tape
ReaD
ReaD Tape
ReaL
ReaL FUNCTION †
STop
SUbroutine
Type INTEGER †
Type REAL †
Type
Write
Write Tape

† These keywords require the total identification of the second word since the same keyword with one component can have a variable name in this position.

fined, a sieve that only checks the significant characters and discards the other characters in the keyword can speed the process. In such a situation invalid keywords can be used without disturbing the compilation. In fact, knowledgeable programmers can concoct their own brand of FORTRAN keywords. For example, one programmer consistently used DAMNITALL as the keyword for DIMENSION.

In FORTRAN IV, where many new statements are added, a major difficulty arises from the inclusion of Hollerith constants in several statements. Thus the characteristic marks of an arithmetic statement, a DO statement and an indexed I/O are no longer unique. For example, the following two statements are, respectively, a legal DO statement and a valid arithmetic (replacement) statement:

```
DO1I = 1, 31, 2
DO1I = 5H,3,10
```

Similarly, the following are a legal IF statement and a valid arithmetic statement:

```
IF(I - 3H)=()1,31,2
IF(I - 3)  = 5H1,3,2
```

Thus if a statement contains the requisite marks, the keyword should also be examined to distinguish between possible statement types. However, the presence of the requisite characters does not determine the type, but merely raises the possibility. If an arithmetic statement containing a Hollerith constant is restricted to be a simple replacement statement, as is recommended, then the DO statement may be distinguished by the following pointers:

(a) Since no constant (and in particular a Hollerith constant containing an = sign) may occur to the left of a replacement sign as in an arithmetic statement or as in the index of a DO statement, then the first = sign must be either the replacement sign in an arithmetic statement or the delimiter between the index variable and the initial parameter in a DO statement.

(b) When the first = sign has been located in a left to right scan, the first variable, function or constant in an arithmetic statement will be delimited by a parenthesis, an arithmetic operator or the end of the statement, whereas the first delimited in a DO statement must be a comma.

An = sign occurring in an IF statement is not mandatory, and thus no rule can determine the first distinguishing mark after the = sign. Thus, if one assumes that a particular statement is an IF statement (after checking the leading two characters) and this turns out to be a false assumption, an attempt can be made to compile the statement as an arithmetic statement. If this fails, the statement is not valid.

Summary

The analysis of nonnatural language statements by the use of parsing to determine the validity and type of statement can be achieved by the use of a generalized parsing routine or by a specific analyzer. The use of a general routine has the advantage that with the input of a language definition, there is no restriction on the language to be analyzed, and a single analyzer can perform the same task for many languages. Further, the intermediate results of the parse, if saved, can serve as indicators and pointers to the compiler generators. However, since parsing with respect to a set of syntax rules (which is akin to an interpretive process) must examine every component of the string to be analyzed, advantage cannot be taken of the inherent properties of each statement type.

The choice of analyzer, therefore, must be determined by the environment into which the compiler is to be placed.

4

SYMTAB—The Symbol Table and Associated Routines

The purpose of a symbol table in a compiler or an assembler is to collect data from the source statements, analyze it for pertinence, assign object-time-storage areas, and to return data to the calling generator for the assembly of the target language instructions. The success of the symbol table routine depends on the information it receives from the calling generator and the type of data that is expected. For example, since the symbol table routine generally examines a source statement without knowing the context, it must handle seemingly similar data in differing manners. In particular, a statement identifier in FORTRAN can easily be mistaken for an integer constant or in PL/I for a variable.

The first task of the symbol table routine must be to extract the elements of the language. Basically, this can be broken into two subheadings: the extraction of (a) numeric constants and (b) symbolic data.

Let us first consider the extraction of integer data from a string of symbols. When any piece of data is to be extracted from a source statement, the first consideration must be given to the delimiting characters of that data. As the source statement is scanned from left to right, the first character will generally give a clue to the type of element in hand (for example, in ALGOL and FORTRAN where a variable name is constrained to begin with an alphabetic character), but the last character of the element cannot be recognized until after it has been passed over and a delimiting character is noted. Since an integer constant may contain only digits, any nonnumeric character will be a delimiting character. However, any leading blanks must be disregarded as must internal blanks. As opposed to an object-time data image, it is normally assumed that internal blanks in a portion of a source statement will be disregarded, though some languages insist on the presence of a blank in certain locations.

The Extraction of Language Elements

If the source statement has been assembled into an area known as CHI, and there exists a word known as CHINXT, which contains the address of the character in CHI currently under consideration, the SYMTAB routine may start with that character and move toward the right. On returning from SYMTAB to the calling routine, CHINXT will contain the address of the delimiting character. Fig. 4.1 shows the extraction process for an integer.

Since this extraction routine can be useful to other routines than SYMTAB, provision must be made for more than simply the collecting string of characters. Similarly, the overzealous transmutation of the string, for example, to the internal representation can detract from the string's general usefulness. The uses to which a simple integer extraction could be put would include:

(a) extracting the logical unit number from a FORTRAN READ or WRITE command,

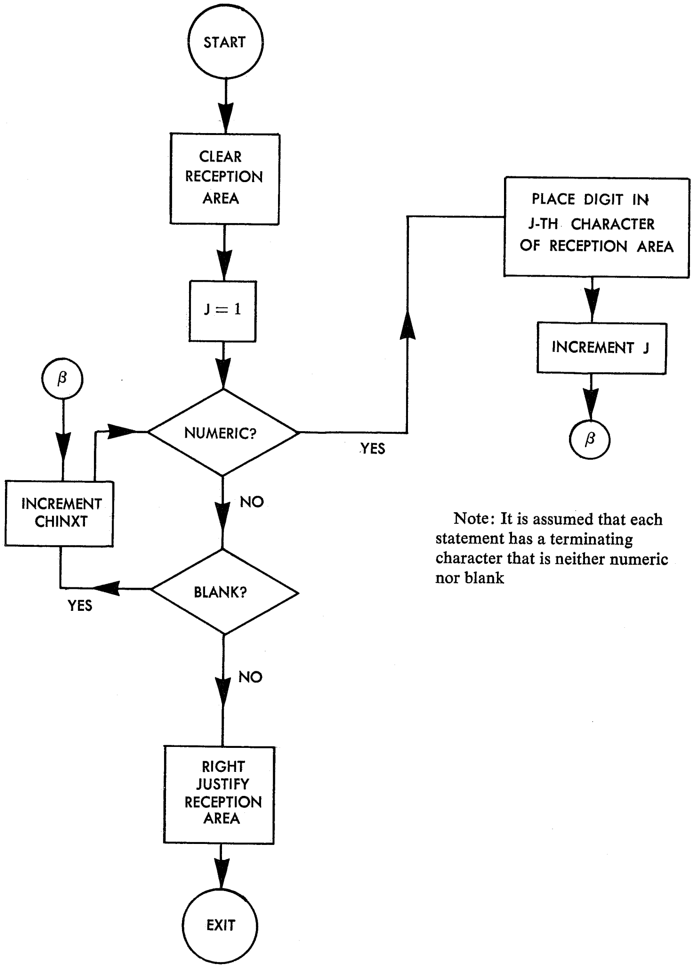
(b) extracting the switch number from an IF(SENSE SWITCH *i*) statement,

(c) extracting the display digits from a PAUSE or STOP statement, and

(d) extracting numeric field widths and other specifications in FORMAT statements.

For example, the specification widths in a FORMAT statement may be extracted by using this routine repeatedly. Further, since neither part (*w* or *d* in an F or E specification) should exceed two digits; if the routine returns the number of digits extracted, then a check for legality can be made.

If the terminating character of an integer extraction routine is H, one may assume that the item is a Hollerith constant. Thus the constant picked up by the routine is to be used by the generator as the count of the char-



Note: It is assumed that each statement has a terminating character that is neither numeric nor blank

FIGURE 4.1 Integer Extraction Routine (INTEX)

acters to be stored as the Hollerith constant. The characters to be extracted can then be extracted irrespective of their value. A constant such as 5H,1,38 would be extracted as ,1,38.

The extraction of a decimal number (or REAL number in FORTRAN) involves more testing than for the extraction of an integer number since the number of valid forms is greater. Further, the internal representation of a real number is not a simple rearrangement of the source data, but a collection of three parts: the integer part, the fractional part, and the exponent. In a decimal machine, these parts may be collected simultaneously, whereas in a binary machine, they must be kept separate in BCD form until a conversion from external to internal mode can be accomplished. However, if the compiler has a target language, such as an assembly code that accepts numeric data in external form, the task of conversion may be either postponed or handled by some other system. We shall consider the following techniques:

(a) source data to decimal internal real mode with a biased exponent and left-justified mantissa,

(b) source data to integer part, fractional part and exponent in external mode, and

(c) external mode to internal binary.

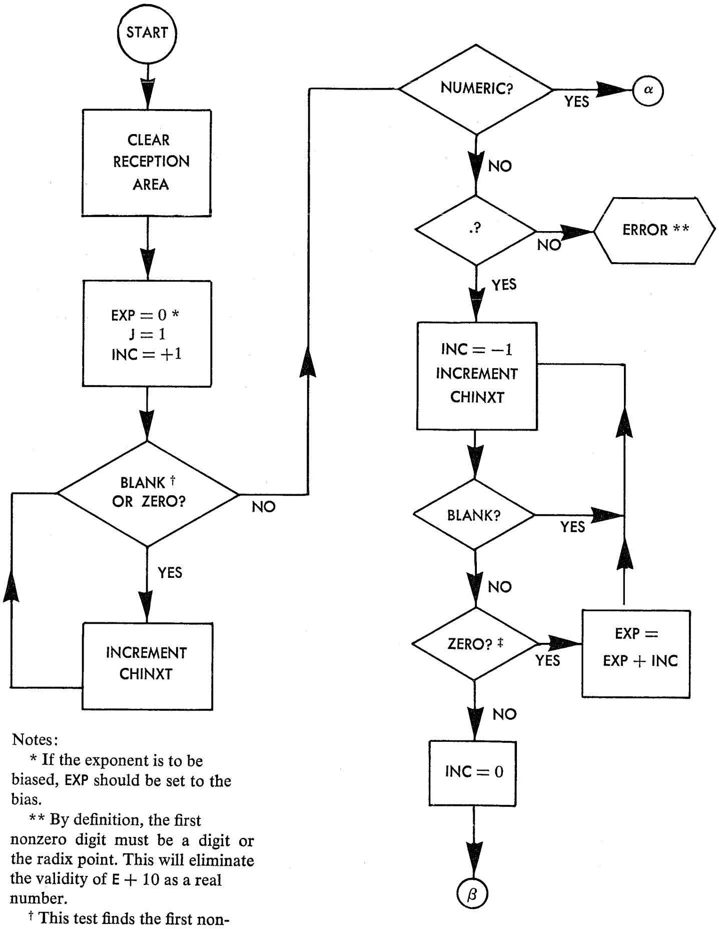
First consider that a real number may be defined as:

<simple real no> := *<integer>*.|*<integer>*.*<integer>*|.*<integer>*
<exponent no> := *<simple real no>*E*<sign>**<integer>*|
*<integer>*E*<sign>**<integer>*
<real no> := *<simple real no>*|*<exponent no>*

The significant characters in a real number are

. (radix point)
E (the symbol for *"times 10 to the power"*)

and the right delimiting character, which in an assignment statement will be either an operator or the end of the statement. In a logical IF statement the delimiter may be either an operator or a right parenthesis, or even a second decimal point (as in IF(0.0.NE.X) . . .). In a DATA statement list the delimiter may be a comma.



Notes:

* If the exponent is to be biased, EXP should be set to the bias.

** By definition, the first nonzero digit must be a digit or the radix point. This will eliminate the validity of E + 10 as a real number.

† This test finds the first nonzero nonblank character.

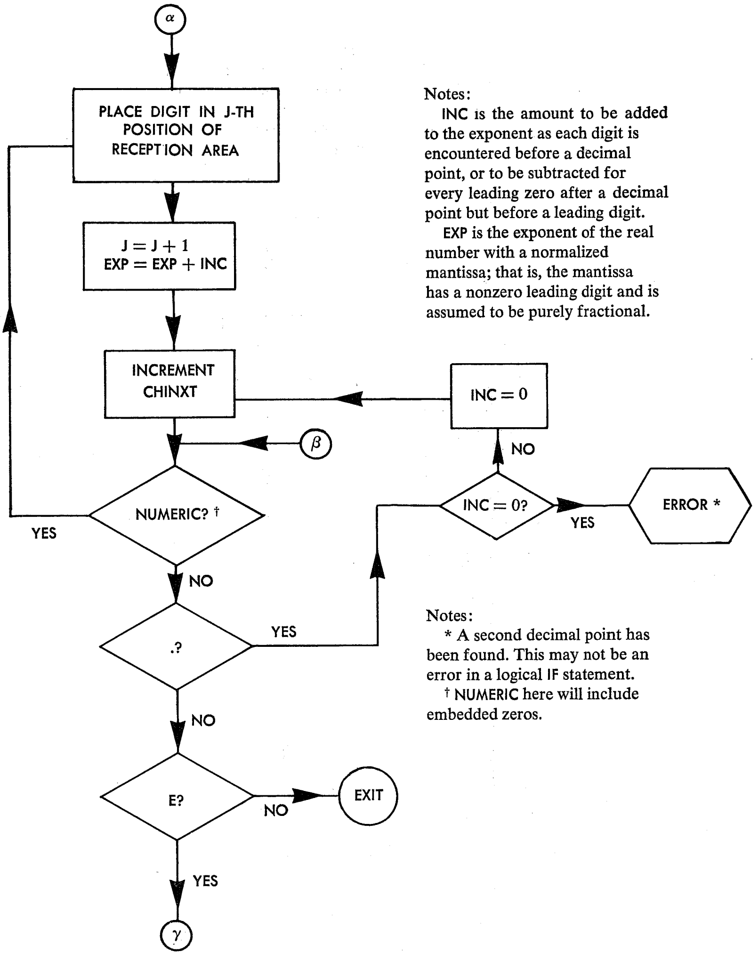
‡This test takes care of zeros before the leading digit but after the radix point.

FIGURE 4.2 Real Number Extraction Routine

In extracting a real number from a source statement, the technique of handling a zero must be considered in one of three manners:

1. After a leading nonzero digit, it should be treated as a digit.

2. Before a leading nonzero digit and prior to the radix point, it should be ignored.



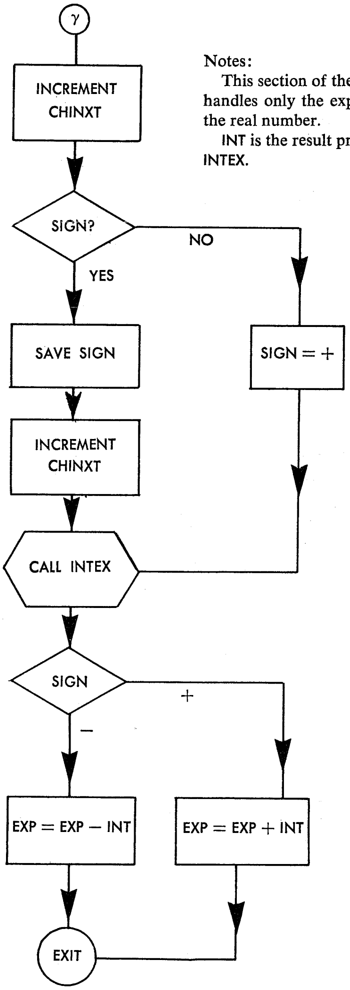
Notes:
INC is the amount to be added to the exponent as each digit is encountered before a decimal point, or to be subtracted for every leading zero after a decimal point but before a leading digit.
EXP is the exponent of the real number with a normalized mantissa; that is, the mantissa has a nonzero leading digit and is assumed to be purely fractional.

Notes:
* A second decimal point has been found. This may not be an error in a logical IF statement.
† NUMERIC here will include embedded zeros.

FIGURE 4.2 (continued)

3. Before a leading nonzero digit but following the radix point, it should be ignored, but with an adjustment of the exponent.

Figure 4.2 charts the routine for extracting a real number from a source statement and converting it to internal decimal real mode. For the purpose of extracting a real number and maintaining the separate identities of the three parts, an alternative routine may be organized that utilizes the integer extraction routine. This is shown in Fig. 4.3.

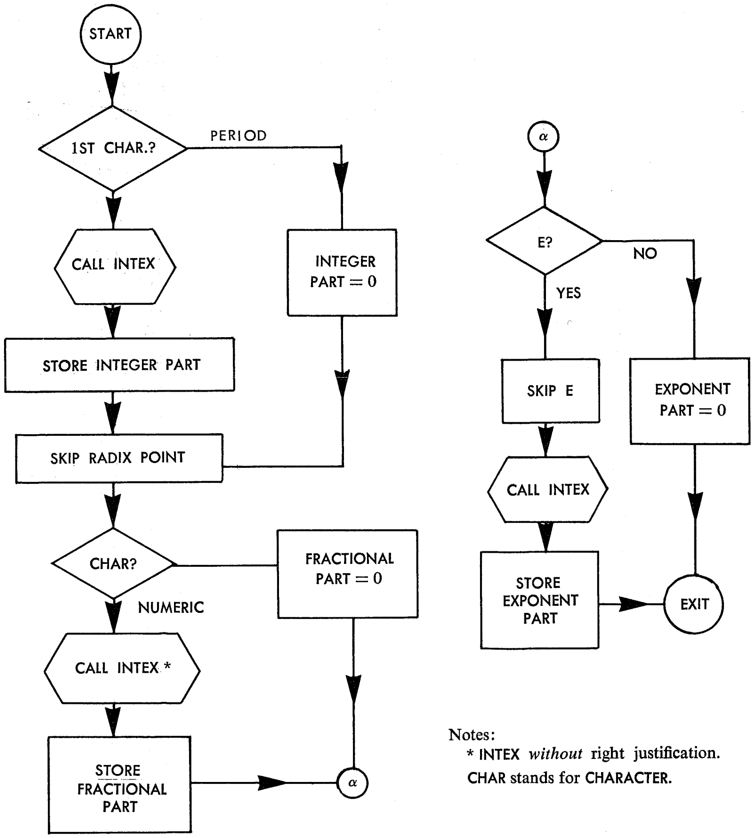


Notes:
This section of the routine handles only the exponent part of the real number.
INT is the result provided by INTEX.

FIGURE 4.2 (continued)

These extraction routines have merely extracted numbers from a string of characters in the source statement and stored these numbers in their unadulterated form in a set of special bins. However, the form in which the data are read into the computer from the source document (tape, card, etc.) is not, generally, the form in which the information is to be manipulated at object time. For example, in binary machines the digits (0–9) can be represented as

Decimal	*Binary*
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001



Notes:
* INTEX *without* right justification.
CHAR stands for CHARACTER.

FIGURE 4.3 Real Number Extraction without Conversion

However, multidigit decimal numbers cannot be represented by the direct translation of digits according to the above table. For example, the decimal number 14_{10} does not code directly to 00010100_2, but rather to 00001110_2. To restrict the internal code of a character to merely 4 bits will restrict the

TABLE 4.1 INTERNAL BINARY REPRESENTATIONS OF EXTERNAL CHARACTERS

External Character	*Internal Octal*	*Internal Binary*
A	21	010001
B	22	010010
C	23	010011
D	24	010100
E	25	010101
F	26	010110
G	27	010111
H	30	011000
I	31	011001
J	41	100001
K	42	100010
L	43	100011
M	44	100100
N	45	100101
O	46	100110
P	47	100111
Q	50	101000
R	51	101001
S	62	110010
T	63	110011
U	64	110100
V	65	110101
W	66	110110
X	67	110111
Y	70	111000
Z	71	111001
0	00	000000
1	01	000001
2	02	000010
3	03	000011
4	04	000100
5	05	000101
6	06	000110
7	07	000111
8	10	001000
9	11	001001

allowable number of characters to 16, whereas in general we will wish to represent at least 48, or even 64, characters. For programming ease, most computers are arranged so that the memory words may be broken up into octal digits since any binary number can be converted to an octal number if the bits are merely grouped into sets of three, to the left and right of the radix point. That is, $011{,}010{,}110.101{,}110{,}110_2$ is equivalent to 326.566_8. Thus if three bits is the basic unit (byte) of information in the computer, two bytes or six bits will be a logical group to represent the whole character set of the machine, as in the set shown in Table 4.1.

Conversion of Numeric Values to Internal Mode

If one possesses a computer that stores its data in purely decimal form and that performs decimal arithmetic, the conversion from decimal to binary mode can be expressed as a simple algorithm. However, if one possesses such a machine, the need to convert from decimal to binary is of less importance than with other computers. As an aid to the consideration of conversion techniques, let us first consider the conversion from binary to decimal mode. Since both number systems are positional, a bit in a given position (with relation to the radix point) can be translated to its equivalent decimal number. For example, the presence of a bit three places to the left of the radix point may be converted to 4_{10}; a bit in the sixth place may be converted to the left to 32_{10}; and a bit in the third position to the right may be converted to 0.125_{10}. Thus one may translate a binary number to decimal mode by looking the positions of the bits up in a table and adding, in decimal mode, the results found in the table. For example, 10110111_2 is equivalent to $1\times2^7 + 0\times2^6 + 1\times2^5 + 1\times2^4 + 0\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0 = 128 + 0+32 + 16 + 0 + 4 + 2 + 1 = 183_{10}$.

Problem

4.1 Convert the following binary numbers to decimal mode:

(a) 110110	(c) 011011
(b) 101010	(d) 100100

By a similar technique, decimal numbers may be converted to binary mode: Very simply, find the largest decimal number that is an integer power of 2 and is smaller than the number to be converted, place a unit in the binary field in the equivalent position, subtract the power of 2 from the original decimal number, and repeat on the remainder until it is reduced to zero.

Consider the decimal number 173_{10}:

$$\begin{array}{rl} 173 & \\ \underline{-128} & = 1\times 2^7 \\ 45 & \\ \underline{-32} & = 1\times 2^5 \\ 13 & \\ \underline{-8} & = 1\times 2^3 \\ 5 & \\ \underline{-4} & = 1\times 2^2 \\ 1 & \\ \underline{-1} & = 1\times 2^0 \\ 0 & \end{array}$$

Thus the binary equivalent of 173_{10} is $1\times 2^7 + 1\times 2^5 + 1\times 2^3 + 1\times 2^2 + 1\times 2^0$, or in positional form, 10101101_2. Although this conversion process is correct, it is somewhat clumsy.

Expanding the positional notation of a binary number to its power series, one may write a binary number in the nested form:

$$((\ldots((b_n\times 2 + b_{n-1})\times 2 + b_{n-2})\times 2 + \ldots b_2)\times 2 + b_1)\times 2 + b_0$$

where b_i is the $(i+1)$th digit to the left of the radix point.
Now if D is the decimal number to be converted into binary mode, one can write:

$$D = ((\ldots((b_n\times 2 + b_{n-1})\times 2 + b_{n-2})\times 2 + \ldots b_2)\times 2 + b_1)\times 2 + b_0$$

Dividing both sides of this equality by 2 in integer mode gives:

$$D' + R = (\ldots((b_n\times 2 + b_{n-1})\times 2 + b_{n-2})\times 2 + \ldots b_2)\times 2 + b_1 + b_0$$

where D' is the integer quotient of the left-hand side, and R is the remainder, which obviously will be either 1 or 0. The remainder of the division of the left-hand side must be the same as the remainder on the right-hand side, that is, b_0. Thus the low-order digit of the binary equivalent may be determined by a simple division by 2 in decimal mode. The quotient of the division (D') is equivalent to the higher order bits in the binary number, and thus continued division and extraction of the remainder will produce the whole binary equivalent. Such a process is shown in Fig. 4.4.

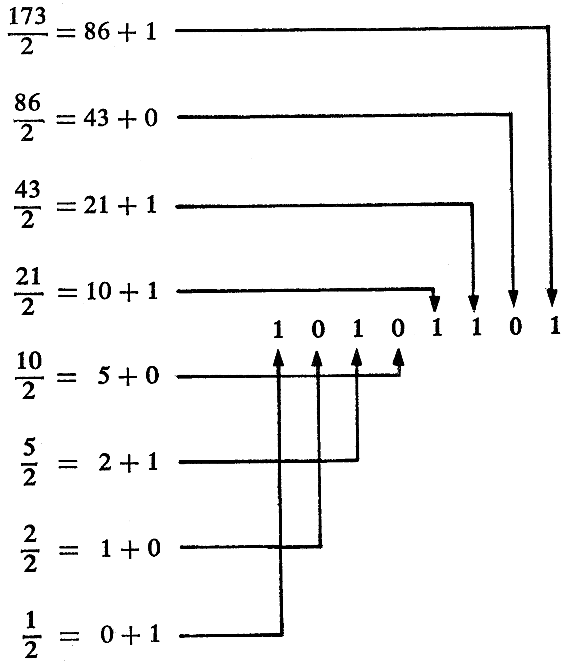


FIGURE 4.4 Decimal to Binary Conversion by Division by Two

While this technique seems simple and straightforward, it can only be used with a decimal machine since the division by 2 of decimal numbers must entail the borrowing of tens. Further, in a binary machine the internal representation is not a correct representation of the decimal number. For example, consider a 4-bit representation of digits. The source language internal representation of 173_{10} will be 000101110011_{bcd}. Now dividing by 2 is equivalent to a shift to the right of one place, so dividing the *BCD* representation by 2 gives a quotient of 00010111001_{bcd}, with a remainder of 1 (which is shifted off). This quotient is equivalent to the number

$$0(11)9_{10}$$

where the middle digit is greater than 10. In reality, a 4-bit internal representation can only truly represent a hexadecimal number (base 16) in which a division by 2 would produce the correct quotient and remainder. Thus, in terms of the previous algorithm, unless special techniques are used to divide by 2 in decimal mode using hexadecimal or *BCD* data, the algorithm is inapplicable to a binary machine.

Such an algorithm may be developed, though it is extremely tedious and time consuming. For example, a BCD character using six bits to allow the acceptance of all characters in the language will contain at least two leading zeroes which may be made use of in the following algorithm. If an integer number is maintained in BCD input form, then a shift right is almost equivalent to a division by 2, and any bit shifted off the right-hand end is the true remainder. Consider each BCD character to be a separate entity which is shifted right as a whole. Then the movement of a low order bit of an internal byte to the high order position of the next lower byte is an indication of adding five (5) to that lower byte. Thus if we add 5 to each byte in which a high order bit has occurred, and remove the high order bit, we can simulate the borrow feature of a decimal divide. This process is shown in Fig. 4.5.

				Digits Shifted Off
173_{10}	000001	000111	000011	
Shift Right	000000	100011	100001	1
Remove H/O bits and add 5 as appropriate	000000	001000	000110	1
Shift Right	000000	000100	000011	01
No H/O bits occurred	000000	000100	000011	01
Shift Right	000000	000010	000001	101
No H/O bits occurred	000000	000010	000001	101
Shift Right	000000	000001	000000	1101
No H/O bits occurred	000000	000001	000000	1101
Shift Right	000000	000000	100000	01101
Remove H/O bits and add five	000000	000000	000101	01101
3 shifts right	000000	000000	000000	10101101

FIGURE 4.5 Conversion from BCD to Binary Using a Simulated Decimal Division by Two

This technique wastes time, but it does show that decimal division of a BCD string is possible. It must be anticipated that the number of one-bit shifts is equivalent to the word size and that the number of tests after each shift to recognize the existence of high-order bits is equal to the number of BCD characters to be translated. Also, a computer with bit and byte manipulation instructions is necessary.

It should be recognized that each BCD byte is a true representation in binary mode of the decimal digit and *on its own* can be treated as a binary field. Now a decimal number is merely a representation of the true number, the position of each digit implying its power of 10 exponent. That is, 253_{10} is a representation of:

$$2\times10^2 + 5\times10^1 + 3\times10^0$$

If a decimal number takes the form:

$$d_n d_{n-1} d_{n-2} \ldots d_2 d_1 d_0$$

where

$$<d_i> := 0|1|2|3|4|5|6|7|8|9$$

then the number can be written in nested form:

$$(((\ldots((d_n\times10 + d_{n-1})\times10 + d_{n-2})\times10 + \ldots d_2)\times10 + d_1)\times10 + d_0)$$

If each BCD character of the input decimal number is stored in a separate word in memory, then the binary equivalent may be constructed by evaluating the nested polynomial from left to right. For example, the decimal number 104 will be stored in memory in separate words as:

$$\begin{aligned} d_2 &\ 000001 \\ d_1 &\ 000000 \\ d_0 &\ 000100 \end{aligned}$$

The binary equivalent of decimal 10 is 1010_2; so evaluating the polynomial:

$$\begin{aligned} d_2 \times 10 &= 0000001010 \\ + d_1 &= 0000001010 \\ \times 10 &= 00000001100100 \\ + d_0 &= 00000001101000 \end{aligned}$$

that is, 104_{10} is equivalent to 1101000_2. As a check, the result may be converted back to base 10. In terms of base 10 numbers, the binary number is the summation:

$$2^6 + 2^5 + 2^3 = 64 + 32 + 8 = 104_{10}$$

Problem

4.2 Assume that the following decimal numbers are input, so that each digit is stored in a separate word in its BCD equivalent coding, and convert the following to binary mode:

(a) 93	(e) 577
(b) 176	(f) 19
(c) 256	(g) 901
(d) 1000	

We shall not consider the conversion of fractional numbers from decimal (or BCD) to binary since a little internal processing before conversion can always ensure that each number is primarily converted to an integer part and an accompanying exponent. Thus the number can be extracted in integer form, the exponent being supplemented by the appropriate amount. Thus during the extraction of the number from the source document, the initial conversion may be accomplished so that prior to conversion to internal mode (probably binary) the number appears in the form of an integer part and an exponent. However, the external base or radix is not the same as that for the internal storage, and further, it is usually found that mantissas are stored as pure fractions. Therefore although the initial conversion produces an integer and an exponent of, for example, base 10, the subsequent conversion must be to a pure fraction and an exponent of base 2. The conversion from an integer of base 10 to a fraction of base 2 is not difficult. If one converts from the external integer to the internal mode integer, internal conversion to a fraction may be accomplished merely by supplementing the 2's exponent by the amount the radix point is shifted. At this stage of the algorithm, the number 0.104 would be converted through the following steps;

$$0.104 = 104. \times 10^{-3}$$
$$104. \times 10^{-3} = 1101000._2 \times 10^{-3}$$
$$1101000._2 \times 10^{-3} = 0.1101000_2 \times 2^7 \times 10^{-3}$$

One problem still remains: The number now contains two exponents which are not in the same mode. That is, the multiplying factor of base 10 must be converted to base 2 with the appropriate adjustment to the mantissa. Suppose one is given a multiplying factor of 10^n where n is an integer by definition. This is to be converted to the multiplying factor 2^m where m is not necessarily an integer. If

$$10^n = 2^m$$

then

$$\log_2 10^n = \log_2 2^m$$

that is

$$n \log_2 10 = m$$

Now n can be converted to binary by the techniques described above and then multiplied by the constant $\log_2 10$ in binary mode. However, in general m will not be an integer and may be represented in the form $(i + f)$ where i is an integer and f is a fraction. If the original decimal number is written as

$$d \times 10^n$$

then its binary equivalent is $b \times 2^{(i+f)}$, that is, $b \times 2^i \times 2^f$. Thus before the number can be stored, the factor $b \times 2^f$ needs to be evaluated since the exponent must be an integer. This may cause some consternation as one would not expect a compiler to include a routine for such a complex operation as involution unless the operation is a load-and-go one with the compiler, library routines and compiled program resident together. As an alternative, this process may be simulated by a table look up procedure based on the powers of 10.

For example, FORTRAN 3600 (for the CONTROL DATA 3600) contains a table of the integer and fractional exponents in base 2 for the base 10 powers of 1 to 20, 40, 60, . . . 300. With this table, the integer powers of the base 2 exponents may be summed and the fractional parts successively multiplied into the mantissa. Once this task has been completed, the mantissa may be normalized and the exponent adjusted appropriately.

Problem

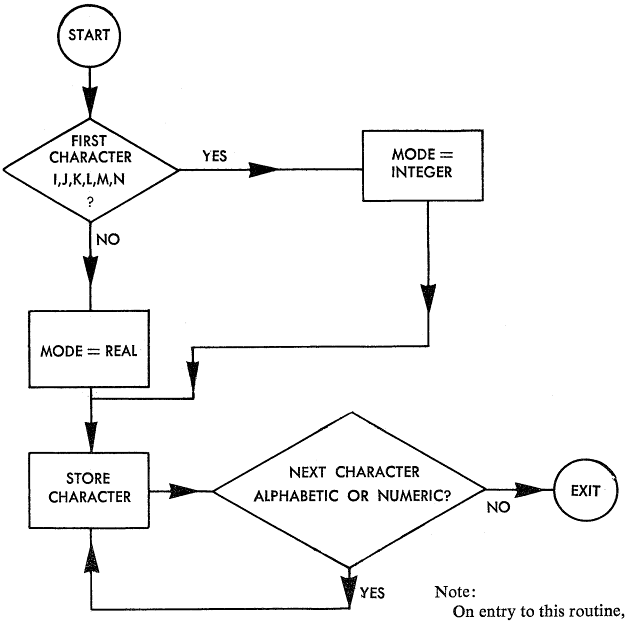
4.3 Using logarithmic tables if necessary, convert the following decimal numbers to normalized binary numbers:

(a) 800.46×10^{20}	(c) -59.381×10^{-45}
(b) 331.24×10^{8}	(d) 3.03×10^{24}

The Extraction of Names

The extraction of variable, function and subroutine names (Fig. 4.6) is by no means as complicated as the extraction of numbers since a conversion between external and internal mode is not required. Terminating characters of names are operators, commas, and periods, the end of the statement and parentheses, depending on context. Of these delimiters, only the opening parenthesis needs special consideration. If the delimiting character of a name is an opening parenthesis, then the name that has been extracted must represent either the name of a subscripted variable, a function reference or a subroutine call, except in the special instances of subprogram definition statements (such as FUNCTION F(X), SUBROUTINE A(X,Y)) or an arithmetic function statement (such as SOMEF(X) = X + 2.*X**2).

When the name has previously occurred in a DIMENSION statement (and therefore is already existent in the symbol table) or has had dimensions


START
FIRST CHARACTER I,J,K,L,M,N ?
YES
MODE = INTEGER
NO
MODE = REAL
STORE CHARACTER
NEXT CHARACTER ALPHABETIC OR NUMERIC?
NO
EXIT
YES

Note:
On entry to this routine, CHINXT is pointing at the primary character of the name. On exit, the pointer is set to the delimiting character.

FIGURE 4.6 ROUTINE VAREXT: The Routine to Extract Variable Names, Function References and Subroutine Calls

assigned to it in a COMMON statement, the name extracted refers to an array or an element of that array. If, in an assignment statement or expression (within an arithmetic IF statement or an expression in a CALL list), and the name has not occurred previously in either a DIMENSION or COMMON statement with dimension, then the name must refer to a function. If the name is a reference to a library function, then the symbol table should have this information; the absence of such information would lead the compiler to assume that the reference is to a user-defined function. Thus while this may in fact be an error on the part of the programmer who has omitted to list this array in a DIMENSION statement, the compiler has a logical alternative which masks the error.

Problem

4.4 Write a program that will read a FORTRAN arithmetic statement that does not include Hollerith constants and will output a list of all the variable names occurring in that statement. For example, the input statement

```
AJ1 = B(I+3,J) + 3.0 * (X / (C(K) + 5.0))
```

should output the following list:

```
AJ1   B   I   J   X   C   K
```

Note: The name extraction routine may be written as a subprogram and this routine will be useful in later problems. If a variable occurs more than once in a statement, it will be permitted to output the name more than once.

The Data in the Symbol Table

The compile time symbol table contains information pertinent to the five types of data that appear within the source language, that is, simple variables, dimension variables and arrays, statement numbers, subprogram names, and constants. Together with this information, which acts as the key to table, appears data necessary to the compilation of each statement. For example, the following table lists some of the information required by the various generators:

Variables: Mode? Real, integer or complex.
Dimensioned? If so, what are the dimensions?
Did the variable name appear as a formal parameter?
Did the variable appear in a COMMON statement or was it forced into COMMON by way of an EQUIVALENCE statement?
Has the variable been defined within the program; that is, is there a statement which will assign a value to this variable?

In the case of a compiler without the intermediate stage of translation to an assembly language, that is, a one- or two-pass system, what is the object time address of this variable?

Functions: Is it a library function defined as part of the system or is it a user-defined function?
Is it a global function (that is, is it available to all subprograms except itself) or is it local (that is, is it defined in an arithmetic statement function)?
What is the mode of the function; that is, what is the mode of the result of the function?
What is (or are) the mode(s) of the argument(s)?

Since the key to each entry in the symbol table is the item itself stored in internal mode (that is, BCD for names and binary for constants), the recovery of any item together with its relevant information can be severely impeded by the necessity to compare the item in hand with every other item. Therefore to speed this search, each entry should be equipped with another secondary key that will describe the type of information being stored. For example, if one knows that the item in hand is a constant and the object time stored location needs to be determined from the table, it will be pointless to compare the item in hand with the variable names in the table. Furthermore, there is possibility that a constant stored in binary mode (in a one-pass system) will have a pattern of bits that is identical to the BCD representation of a name. The presence of the secondary key will obviate the possibility of a confusion arising in this case.

Conditions that Define a Quantity

Part of the task of the compiler should be to give diagnostics to the programmer to indicate that a program is, for example, in error as a result of a variable being undefined within the source document. Thus the attempted execution of this program may fail due to the absence of a meaningful value for this variable. Similarly, an undefined statement number can cause both the compiler and the object program considerable difficulty in attempting to execute a GO TO statement.

Variables are defined by:

1. Appearance in a COMMON statement. This is not a foolproof test for definition, but when programs are to be overlaid, the compiler cannot check for the presence of a value for each variable.

2. Appearance in an EQUIVALENCE statement. However, if a set of lists describing the variables that have appeared in EQUIVALENCE groups is maintained during compilation, the possibility of undefined equivalenced variables may be checked further.

3. Appearance as an argument in a CALL statement. This also is not a true test of definition, since there is no way to determine within the subprogram which elements are to be used as input to the subroutine and which are to be given values by the routine.

4. Appearance as a formal parameter (dummy argument) in the defining statement of a subprogram.

5. Appearance on the left-hand side of an assignment statement.

6. Appearance as the second variable in an ASSIGN statement, that is, as the variable to which the statement number is assigned; for example, in the statement

```
ASSIGN 19 TO K
```

K would be taken to be defined.

7. Appearance in a DATA statement.

8. Appearance as an element of an input statement.

None of these tests for value definition are completely foolproof since only one (the appearance in a DATA statement) assures that a variable is assigned a value before it is to be used as a source of information and the assignment of a value to an element of an array (even in the DATA statement) does not ensure that all other elements have values. However, if none of the above conditions applies, then the compiler may emphatically give a diagnostic warning. In certain instances, this should be disastrous enough to prevent execution, whereas in others the programmer is merely given due warning of impending failure.

9. As a special case, the index parameter of a DO statement is taken to be defined within the range of the DO. Although it would appear that the index parameter (or control variable) of a DO loop is defined within the range of the DO and will take on the value of the last passage through the loop plus the increment, except when the range is exited abnormally, the specifications for FORTRAN state (with regard to the conditions which terminate the repeated execution of a DO range)† :

† Sec. 7.1.2.8, USASI Standard FORTRAN.

> *If the value of the control variable is greater than the value represented by its associated terminal parameter, the* DO *is said to have been satisfied and* the control variable becomes undefined.

This situation results from the original implementations of the FORTRAN compiler. In these versions the DO range was controlled from a set of index registers in which the various parameters were stored. However, because of the limited number of registers it was necessary to reuse registers outside the range, and thus there was no guarantee as to the current value of the index parameter (or control variable). Thus rather than leave the value of the parameter to the wiles of the compiler writer, it was felt better to state that the value of the parameter became undefined. Further, since the value of the parameter was stored in a register instead of direct access memory, its value was not available without considerable programming difficulty and subsequent loss of computer time.

10. Statement numbers are defined by appearance in columns 1–5 of a source statement.

11. Subprogram names are defined by appearance in a FUNCTION, SUBROUTINE or PROGRAM statement, by appearance on the left-hand side of an arithmetic statement function, or by implication as a standard library subprogram.

12. Constants are always defined.

Organizing the Symbol Table

The purpose of the compile time symbol table is to store the elements of the program together with information pertinent to the uses to which each element has been put. Since object time addresses of data are not subject to inspection at object time, these addresses may be assigned in any order which is convenient to the programmer or the computer. However, the compile time symbol table is constantly being scrutinized and thus its construction must be amenable to this inspection. The SYMTAB routine which takes this task upon itself, may be called upon at any time to perform one of the following jobs:

(a) Post an item and its associated data.

(b) Retrieve the data associated with any item.

(c) Delete an item and its associated data.

All these activities involve the searching of the table to locate the item or to recognize the absence of that item, and hence the efficiency of this search affects the efficiency of the whole compiler.

The obvious technique of arranging the symbol table is to divide the memory into a distinct number of cells and to assign each entry to the next available cell in the table vector. To retrieve the data associated with any item, or to locate an item for deletion, the in hand item must be checked against the table from the top down. Thus to retrieve an item, the average number of comparisons will be equal to half the size of the present table. Similarly, determining that an item is present requires the same number of checks.

If certain names in the language are to be reserved for special purposes, then the number of comparisons will be increased by this number each time. Thus the search time in a compiler with a set of reserved names can increase substantially, compared to one without such restrictions.

An alternative manner of symbol table organization † is that which was originally used in the SOAP assembler for the IBM 650. Instead of assigning compile time cells to an item in the order of their presentation to the symbol table, this technique uses the internal representation of the item as input data to a routine that creates an address within the available symbol table area. This address is then checked to verify the existence of the item. If the location is blank, the item does not exist in the table and may be posted at that position. If the location is already in use, the item that is stored and the item in hand must be compared. If they match, the search is complete. If they are not equal, then a further search must be made. This may be done by one of two techniques: Either a sequential search can be initiated or a link can be formed to a separate list ordered by appearance of the item. The former technique has two advantages: Only a single table is required, and if a blank cell is located during the sequential search, then the item in hand does not have a match within the table since an entry would have had to take the same route. However, when the table is reaching the saturation point, the number of comparisons either to locate the matching item or determine the absence of that item approaches that of a standard sequential search. Further, the sequential search must be so organized that the table is considered to be cyclic. That is, since the starting point for the sequential search is not the first item in the list, special

† J. Field, D. A. Jardine, E. S. Lee, J. A. N. Lee and D. Robinson, *Kingston FORTRAN II,* 1620 Users Group Conference, Chicago, 1964; also A. Batson, "The Organization of Symbol Tables," Comm. ACM, Vol. 8, No. 2, 1965.

arrangements must be made to cycle from the last item in the physical table to the first item in the table and also to stop if the cycle continues up to the item onto which the name of the item was originally mapped. If the latter situation occurs, then the table is full and the item has not been included in the table.

A method by which the primary table is linked into a secondary table requires that each entry be supplemented by an address defining the location of the next entry in this string. Thus if a match is not made in the primary table, the link address is checked to determine whether there is a list emanating from that point. If the address is blank, then such a list is not present and a match will never be accomplished. Thus the item may be posted at the next available cell in the secondary table. If a link address exists, then the item at that address must be checked, and if a match is made, then the search is complete. If not, the link address at this location is to be tested, and the search will continue through the secondary table. The checking of reserved words in a random search technique does not detract from the overall efficiency of the system since a single mapping of

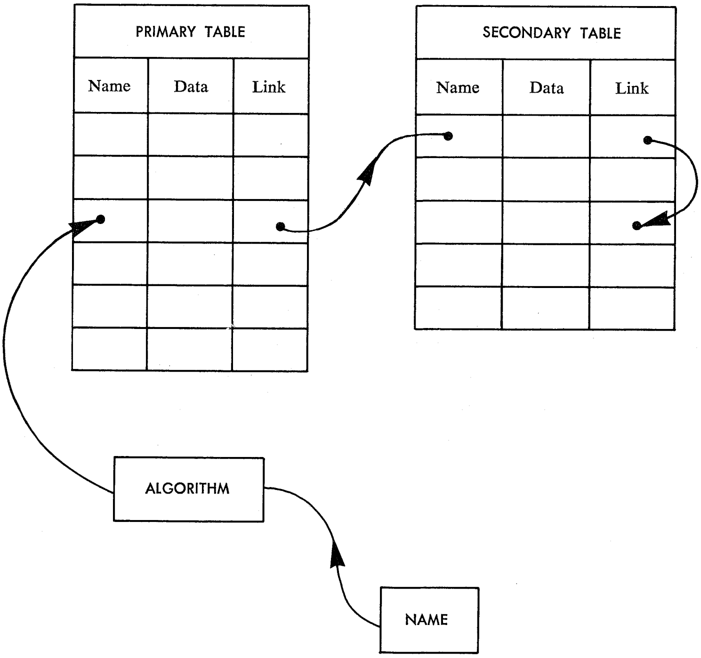


FIGURE 4.7

the word into the primary table will suffice to determine whether the item is reserved. Thus in the worst case, only one cell must be checked to determine that an item is not a reserved word as opposed to the necessity to check every reserved word in a sequential search. The linkages of this pseudo-random addressing technique are shown in Fig. 4.7.

This technique is much faster than the sequential search system, provided the table is not approaching the saturation point. Table 4.2 shows some experimental results.

TABLE 4.2 [a]

No. of Items in Table	*Sequential Search*	*"Random" Search*
20	40	2.14
50	55	2.25
100	80	2.37
200	130	3.14
To locate a reserved word	15	1.08 [b]

Retrieval time: τ/T [c]
Table size: 400 cells
No. of reserved words: 30

[a] From A. Batson "The Organization of Symbol Tables," Comm. ACM, Vol. 8, No. 2, 1965.

[b] This time is not exactly 1.00 since some reserved words mapped into the same location in the primary table and thus had to be stored in the secondary table.

[c] T is the time to examine a single entry in the table; τ is the total search time.

This "random" posting and retrieval technique affords substantial time-saving advantages but also has some inherent disadvantages:

(a) The tables must be arranged so as to be cyclic.

(b) Although it is obvious when the secondary table is full in a linked secondary system since items are posted sequentially, vacant cells may be present in the primary table, and only a sequential search through the primary table will reveal these cells. Thus the table may appear to be full when the secondary table becomes full, and only special action can determine the validity of this assumption.

(c) If the programmer requires a symbol table mapping, the table together with any imbedded blanks must be searched, whereas in a sequential table the table is dense and can be mapped more rapidly.

(d) Since all variables not in COMMON are local to a subprogram, generally these variables must be deleted from the symbol table when an END statement is located. This prevents confusion between similarly named variables in two subprograms and allows the omission of the subprogram name from the data associated with each item. Further, the deletion of items from the symbol table after the compilation of one subprogram will leave room for the new items in the next subprogram. Thus the total size of the table need not be excessive. Even though the variables that are in COMMON define an area that is common to all subprograms, the names and pertinent data need not be remembered. Only the bounds of the COMMON area are relevant to succeeding subprograms, and thus more space is available for the next set of postings. Function names and constants are common to all subprograms and therefore are not to be deleted on the location of an END statement.

However, all these deletion processes suffer from the same problems found in forming a map of the table.

The pseudo-random technique of symbol table organization together with a secondary or even a tertiary table has the advantage that where there is variable amount of pertinent data to be stored with each entry the subsequent table can be used for these variable data strings, whereas the primary table(s) which contain the keys and fixed-length names may be of standard cell sizes. For example, a dimensioned variable must be stored in the symbol table not only by name and type but also with the dimensions. Thus the name may be stored in the primary table (if space exists) together with a link to a subsequent table in which the dimensioning information is stored. Thus if a single word is reserved for the storage of the next available address in the subsequent table, variable numbers of words may be reserved for the dimensioning information without upsetting the whole organization of the symbol table and at the same time only using space as required without leaving blanks in cells that were originally set at a size equal to the maximum required for any string of data.

Such a technique can be utilized for the construction of the symbol and occurrence table of an indexer.† An indexer is a program, the input to which is the source language of, for example, an assembly system, and the output of which is a table of the statements in which each variable is either used or defined. For example, the following COMPASS (Control Data

† R. Pratt, *Referencer and Indexer.* IBM 1620 Program Library No. 1.1.014.

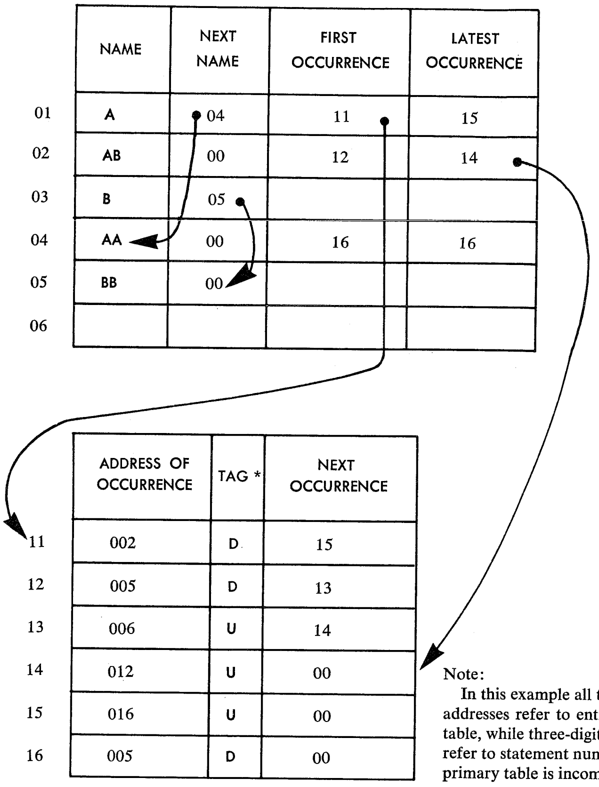
3600) listing includes the names of 10 variables, some of which are actually defined in this portion by their appearance in the first eight character positions. Thus the output of the indexer for this partial program will show that some variables are undefined.

```
        STA      CACHE          001
A       IF,EQ    CACHE,STORE    002
        ENI      99,1           003
B       IFU                     004
AB      RAO      AA,1           005
        IJP      AB,1           006
        ENI      49,2           007
C       IFU                     008
        ENDIF    B              009
        ENI      49,2           010
AB      RAO      AA,1           011
        IJP      AB,1           012
        ENDIF    C              013
BC      RSO      BB,2           014
        IJP      BC,2           015
        ENDIF    A              016
        LDA      NEXT           017
```

The indexer output for this program would be:

```
002    A       016
       AA      005    011
005    AB      006    012
004    B       009
011    BB      014
014    BC      015
008    C       013
       CACHE   001    002
       NEXT    017
       STORE   002
```

where the first column of figures denotes the statement in which the variable is defined, and the figures to the right of the variable denote the statements in which that variable name appears. Thus the variable AB is defined



	NAME	NEXT NAME	FIRST OCCURRENCE	LATEST OCCURRENCE
01	A	04	11	15
02	AB	00	12	14
03	B	05		
04	AA	00	16	16
05	BB	00		
06				

	ADDRESS OF OCCURRENCE	TAG *	NEXT OCCURRENCE
11	002	D	15
12	005	D	13
13	006	U	14
14	012	U	00
15	016	U	00
16	005	D	00

* D = definition
U = a use

Note:
In this example all two-digit addresses refer to entries in either table, while three-digit addresses refer to statement numbers. The primary table is incomplete.

FIGURE 4.8

at statement number 005 and used in the statements numbered 006 and 012. Similarly, the variables AA, CACHE, NEXT and STORE are not defined.

Such a table is extremely useful in the debugging or updating of programs, for the change in action or effect of a single variable at several points in the program or the execution of a program may require the checking of each occurrence of that variable. Similarly, in reading a program, the index is an aid to the tracing of the logical flow of the execution of the program.

When a program is prepared to perform the task of an indexer, both the number of references to each variable and the total number of variables are unknown. Thus given a sufficiently large portion of memory, the random technique, with a subsequent table for the storage of the references, is a suitable system. The primary table may be used to post the name of the variable and three addresses:

1. The address of the storage location of the next variable which maps into the same position in the primary table.

2. The address of the word in which the occurrence of the first use of the variable is noted.

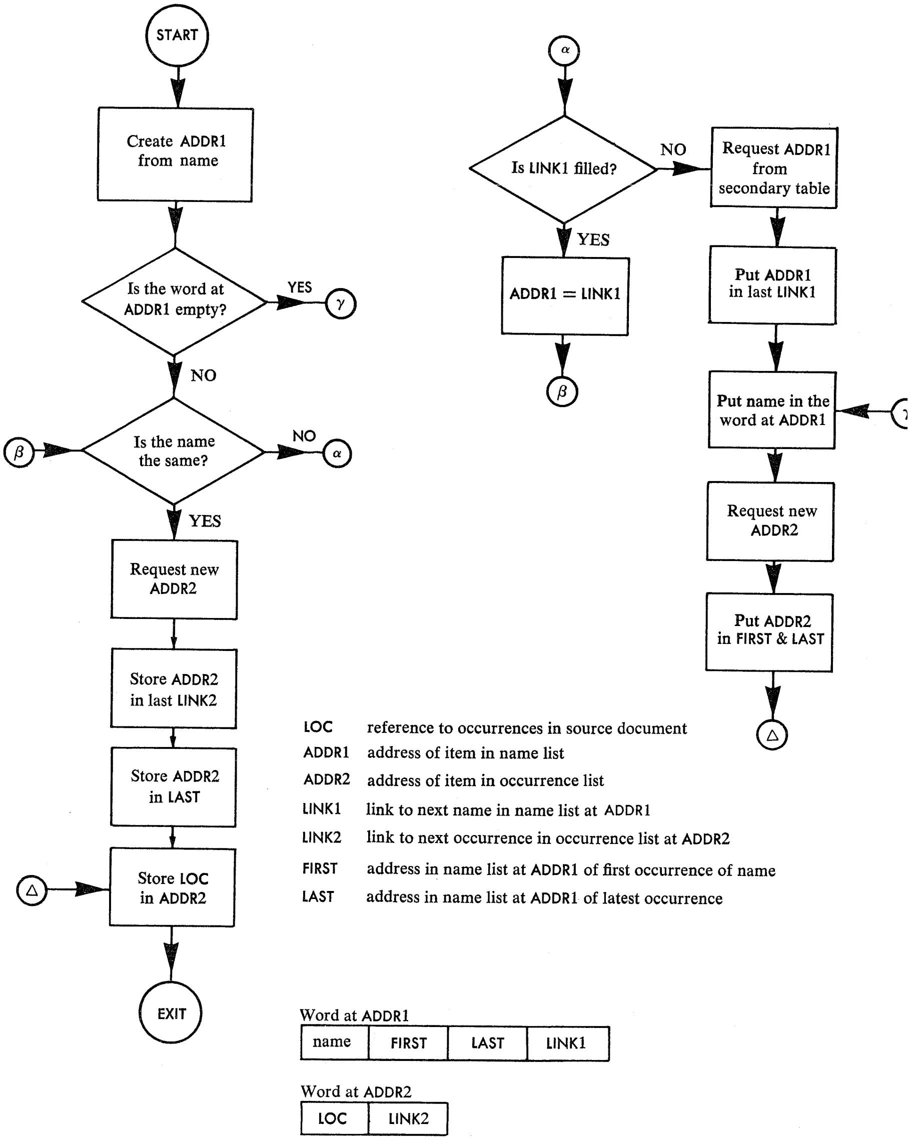
3. The address of the last word in the subsequent table at which the number of the statement in which the last (or latest) occurrence has been noted.

Within the occurrence list associated with each variable will be: (a) the number of the statement at which the variable was found, (b) a tag to indicate whether the occurrence was merely a use or a definition and (c) an address linking this posting with the word in which the next occurrence is noted. This system is shown graphically in Fig. 4.8. The address of the last (or latest) occurrence in the primary table is merely a luxury which enables the posting or retrieval routine to locate the last entry without chaining through all previous occurrence postings. This address is continually updated as new postings are made. The posting routine is shown in Fig. 4.9.

Problem

4.5 Literary critics have taken to using a computer to automate many of the techniques of literary analysis and, as a result, have been able to authenticate the authors of several important works. For example, using a computer to analyze the metric form of the *Odyssey,* researchers concluded that the whole work was written by one person, and since there is other evidence to ascribe certain portions to Homer, the evidence as a whole leads to but one conclusion. Similarly, the authorship of the *Federalist Papers* was determined by an examination of the frequency of such key words as "upon," "while" and "whilst." The technique of word counting and listing all occurrences in context is known as forming a concordance. Such a program involves the reading of a text in machine-readable form (cards, tape etc.), extracting each word and keeping track of each occurrence of that word.

Assume that a text has been prepared on cards and may contain punctuation. Write a program to extract the words, to count the number of occurrences and, eventually to output both an alphabetized listing of the words and counts and a second listing in the order of frequency of occurrence, commencing with those



LOC	reference to occurrences in source document
ADDR1	address of item in name list
ADDR2	address of item in occurrence list
LINK1	link to next name in name list at ADDR1
LINK2	link to next occurrence in occurrence list at ADDR2
FIRST	address in name list at ADDR1 of first occurrence of name
LAST	address in name list at ADDR1 of latest occurrence

Word at ADDR1

name	FIRST	LAST	LINK1

Word at ADDR2

LOC	LINK2

FIGURE 4.9. OCCURRENCE POSTING: Input Parameters include Name and Location of Occurrence

words of least frequency. Do not overlook the fact that some of the text words may not fit into a single computer word.

For this particular example, the following symbolism should be used to represent those characters that do not exist in a standard 48 character set:

!	exclamation mark	/.
?	question mark	@.
:	colon	..
;	semicolon	.,
‘	quote and unquote	*
’	apostrophe	/
-	hyphen	- (minus sign)
—	dash	-- (repeated minus)

Note the following: (a) Poets and writers of prose may use shortened words such as ’twas or ’t. The apostrophes should be included as part of these words. (b) A hyphenated word should be regarded as a single word. (c) When the last character in a sentence is an apostrophe, the text should be prepared so that there is a blank between this character and the period. In this manner there can be no confusion between this combination and an exclamation mark.[†]

Tree Structures

The technique of retrieving or posting items in the secondary table of a pseudo-random symbol table wherein successive items in the table are linked to each other by each entry containing the address of the next entry may be used as the fundamental posting and retrieval system. However, starting from a single entry and proceeding to later entries by means of links is no more efficient than the sequential system. Similarly, in a pseudo-random technique, with a large primary table and thus many sublists, the number of items to be examined before determining the existence of the "in hand" item is comparatively small. During this process of comparing the in hand item with the items existent in the table, there are three possible answers: equal, greater than and less than. If an equal condition is found, then the item already exists in the table and the search is completed. However, in an unequal condition, two branches may emanate from that point (or node), one for the *less than* condition and the other for the *greater than* condition. At each succeeding node, the same test may be made and new branches started. Such a structure is known as a *tree.*

[†] For further information see J. A. Painter, *Computer Preparation of a Poetry Concordance,* Comm. ACM, Vol. 3, No. 2, pp, 91–95, 1960.

Trees have been the subject of intensive study [†] for the purposes of informational retrieval systems and thus have been provided with a distinct set of terminology:

Root: The root is the topmost node in the tree. Only one root may appear in a single tree.

Parent: The parent of any node (except the root) is its immediate predecessor in the tree.

Child: The child of a node is one of its immediate successors. Thus if α is the child of β, then β is the parent node of α. With the understanding that a tree is grown from its top down (!) and that each branch of a node either moves downward toward the left or the right, then each node may have only two children (corresponding to the conditions of greater than and less than) known as the left and right children.

Descendant: The descendants of a node are all the nodes that are children, children of children, etc., of that node.

Leaf: A leaf of a tree is a node with no descendants.

Subtree: A subtree is that portion of a tree that contains all the descendants of a single node.

Branch: The branch of a tree is a single path from any given node to any leaf that is one of its descendants.

When information is stored in the form of a tree structure, the accessibility of any single item depends on the length of the longest branch within the tree. For example, an ordered list may be considered as a tree which contains a single branch and within which each parent has only a single child. Thus the retrieval time of any piece of data will be proportionate to the length of the single branch. However, in a tree constructed so that each parent has two children, except where the children are leaves, and the length of all branches does not differ by more than one node, the average retrieval time may be shortened enormously. For example, a tree containing 1,000,000 items may be constructed so that no more than 20 comparisons are necessary either to locate an item or to determine its absence. Thus if a tree contains *n* levels, that is, the length of each branch is *n,* then the maximum number of items in the tree will be 2^n-1, whereas in an unbalanced tree (that is, one in which the branches or subtrees within one

[†] See the references in C. C. Foster, *A Study of* AVL *Trees,* Goodyear Aerospace Corp. Report No. GER-12158, Akron, Ohio, April, 1965; and Proceedings of the A.C.M. Conference, Cleveland, Ohio, August, 1965.

node need not be of equal length), the minimum number of stored items is *n*. The maximum number of items stored in a balanced tree is shown in Table 4.3. Conversely, the same table indicates the maximum number of comparisons necessary to locate a single item in a balanced tree.

TABLE 4.3

n	*Maximum No. of Entries*
1	1
2	3
3	7
4	15
5	31
10	1023
15	32767
20	1048575

The efficient retrieval of information from a balanced tree may also be accomplished by use of a binary search technique on an ordered list of items. That is, given an ordered list the number of comparisons necessary to retrieve an item cannot exceed $\log_2 (m + 1)$, where m is the number of items in the table. A binary search technique compares the in hand item with that at the center of the table and then makes one of three decisions: (a) The item is at that location; (b) the item is above this point (that is, it must occur in the upper portion of the table); or (c) the item lies in the lower half of the table. If the item is not located immediately, then the portion of the table within which it would appear to lie is regarded as a new table and the search procedure repeated. If the table is successively halved until one item remains and that item does not compare equally with that in hand, then the item does not exist within the table.

In a compiler, the order of presentation of variable names and other symbol table data to SYMTAB is purely random, and thus maintaining an ordered list is wasteful of time. Further, the unrestricted formation of a tree will not necessarily produce a balanced tree from which data may be extracted rapidly. Foster † has shown that trees may be kept balanced by a simple routine, but the posting and retrieval time, together with that for maintaining balance, far exceeds that for the same procedures in a pseudo-random table.

† C. C. Foster, *Ibid.*

Memory Allocation at Object Time

In a compiler the programmer has very little control over the manner of organization of the object time data tables except with respect to items in COMMON. Thus the compiler must provide the organizational system itself. In fact, the compiler writer could choose to scatter the data over the available memory in a random order similar to that proposed for the compile time symbol table and, provided that the data was readily accessible at object time, the programmer would not suffer because of this process. At the extreme, the individual elements of an array might not be placed in contiguous memory locations but rather in locations convenient to some retrieval system. As a general policy, compiler writers choose to allocate object time data storage in the high-order end of the memory on the basis that generated programs must be placed in sequentially increasing memory locations, subprograms which are relocatable may be fitted into any available space and data must be organized in at least some reasonable fashion. Thus beginning at the highest available location, data are allocated space in descending sequence, arrays being stored backwards † with respect to their storage addresses. The following list describes the needs of each type of variable and constant.

Simple Variables and Constants

1. Real variables are contained in one word, the value being stored as a biased exponent and normalized mantissa.

2. Integer variables are contained in one word, the value being stored in the standard internal integer form.

3. Complex variables are contained in two contiguous words, the value of each part being regarded as a real variable. In effect, a complex variable is treated as a vector of two elements.

4. Boolean variables are contained in one word, or if possible (depending on the computer), as one bit packed into a word with other one-bit variables and constants.

5. All constants are stored in the same manner as their respective variables. However, the value of the constant must be loaded before execution of the object program, whereas locations for variables are unaffected during

† See the footnote on page 206.

loading except when that variable appears in a DATA statement. Hollerith constants are indistinguishable from integer or real constants though, of course, they fulfill a different purpose at object time.

Dimensioned Variables

6. Vectors (one-dimensional arrays) are stored such that their first element is located at the currently highest numbered available address and with succeeding elements at progressively lower-numbered addresses. That is, the vector dimensioned A(10) would be stored in the order A(1), A(2), A(3) . . . A(10) at successively lower memory locations. The address of element A(I) may be calculated from:

Address of A(I) = Address of A(0) − I

where the address of A(0) is called the *base* address of the vector A.

Note that

Address of A(0) = Address of A(1) + 1

7. Matrices (two-dimensional arrays) are stored with the first element, for example, B(1,1), stored at the highest numbered address available. The rest of the elements are stored in columns at progressively lower-numbered addresses. That is, if the first subscript refers to a row position and the second subscript to the column, the order of storage is such that the first subscript changes most rapidly in passing through sequentially lower storage locations:

B(1,1), B(2,1), B(3,1) . . . . .

If the matrix B is dimensioned as B(IMAX,JMAX), then the address of the general element B(I,J) may be computed from the expression:

Address of B(I,J) = Address of B(0,0) − (J*IMAX+I)

where the address of B(0,0) is called the *base* address of B and

Address of B(0,0) = Address of B(1,1) + (IMAX+1)

Note that if the compiler or object time routines do not check for subscripts outside the range of the dimensions declared in the DIMENSION statement, and in particular for zero or negative values of those subscripts, then such subscripts will work properly on the second subscript but not on the first. For example, if B is dimensioned B(3,3), then B(3,1) and B(0,2) will refer to the same element, a condition which may be undesirable. That is,

$$\begin{aligned}\text{Address of B}(0,0) &= \text{Address of B}(1,1) + 4\\ \text{Address of B}(3,1) &= \text{Address of B}(1,1) + 4 - (1*3+3)\\ &= \text{Address of B}(1,1) - 2\\ \text{Address of B}(0,2) &= \text{Address of B}(1,1) + 4 - (2*3+0)\\ &= \text{Address of B}(1,1) - 2\end{aligned}$$

However, the location of B(2,0) will be two words higher than B(1,1). Since zero and negative subscripts are extremely useful in the manipulation of data that occur in the COMMON area, the facility to refer to elements outside the range of the DIMENSION statement is important. However, the programmer must be warned to know what he is doing and to obtain the storage allocation algorithm. As an example of memory layout, consider the following COMMON statement:

```
COMMON X, A(4), B(2,3)
```

The memory layout would be:

Variable name	*Memory location (base 10)*
X (Also Base of A)	3999 (for example)
A(1)	3998
A(2) (Also Base of B)	3997
A(3)	3996
A(4)	3995
B(1,1)	3994
B(2,1)	3993
B(1,2)	3992
B(2,2)	3991
B(1,3)	3990
B(2,3)	3989

8. Three and higher dimensional arrays are stored in the same manner as simpler arrays; thus elements are stored at progressively lower-numbered addresses with the first subscript of the array varying most rapidly and the final subscript least rapidly. For example, an array with dimensions A(2,3,3,2) is stored in the following order (from highest numbered memory address to lowest address):

A(1, 1, 1, 1)	Highest Numbered Address
A(2, 1, 1, 1)	
A(1, 2, 1, 1)	
A(2, 2, 1, 1)	
A(1, 3, 1, 1)	
A(2, 3, 1, 1)	
A(1, 1, 2, 1)	
A(2, 1, 2, 1)	
A(1, 2, 2, 1)	
A(2, 2, 2, 1)	
A(1, 3, 2, 1)	
A(2, 3, 2, 1)	
A(1, 1, 3, 1)	
A(2, 1, 3, 1)	
A(1, 2, 3, 1)	
A(2, 2, 3, 1)	
A(1, 3, 3, 1)	
A(2, 3, 3, 1)	
A(1, 1, 1, 2)	
A(2, 1, 1, 2)	
A(1, 2, 1, 2)	
A(2, 2, 1, 2)	
A(1, 3, 1, 2)	
A(2, 3, 1, 2)	
A(1, 1, 2, 2)	
A(2, 1, 2, 2)	
A(1, 2, 2, 2)	
A(2, 2, 2, 2)	
A(1, 3, 2, 2)	
A(2, 3, 2, 2)	
A(1, 1, 3, 2)	
A(2, 1, 3, 2)	
A(1, 2, 3, 2)	
A(2, 2, 3, 2)	
A(1, 3, 3, 2)	
A(2, 3, 3, 2)	Lowest Numbered Address

9. *Statement numbers:* In a one-pass system there can be references to a statement number that is not yet defined. In this case, an object time word is needed to allow indirect references to the statement number. The address of the statement can then be filled in when the statement is encountered. After the statement has been compiled, all references to that statement can be compiled directly, and therefore the object time word is no longer needed. The address of the statement may be loaded into the reserved location before execution of the program.

In order to save data storage space at the cost of loading time and at the cost of a larger amount of data emanating from the compiler, a list which describes the location of each reference in the object program to the undefined statement number may be maintained in the compiler and provided to the loader. The loader must then overlay these addresses onto the object program. Although this increases the loader time, the branches may now be executed directly, thus saving execution time.

In a multipass system or even a one-pass system where all references are made after the reference statement has been encountered, no object time storage is required.

10. *Subprogram Entries:* In a system that compiles an object language and data onto an external medium and then reloads for execution, only user-defined and selected library routines need to be loaded, thereby saving both time and memory space. In this situation, library subprograms must be written in a relocatable form and do not occupy a fixed location in the memory. Similarly, the location of user-defined subprograms is indeterminate since their lengths are undefined.

A simple method for linking the mainline program to library subprograms which are relocatable and of which there is a predefined number, is to reserve a singly dimensioned array into which, at object time, will be stored the absolute address of the entry point to each subprogram. With this *transfer vector,* links may be made to library subprograms indirectly, each subprogram having a fixed address location in the vector. Thus if there are *n* library routines provided with the system, then the transfer vector should have the same number of entry locations. For example, USASI Basic FORTRAN requires that each compiler be provided with seven external functions: EXP, ALOG, SIN, COS, TANH, SQRT and ATAN; whereas the full FORTRAN must contain 24 basic external functions: EXP, DEXP, CEXP, ALOG, DLOG, CLOG, ALOG10, DLOG10, SIN, DSIN, CSIN, COS, DCOS, CCOS, TANH, SQRT, DSQRT, CSQRT, ATAN, DATAN, ATAN2, DATAN2, DMOD and CABS.

On the other hand the specification lists a number of intrinsic functions (six in Basic FORTRAN, including such functions as ABS and IFIX, and 31 in full FORTRAN), but does not state the number of these which are required in a system. Thus it would appear that the compiler writer can choose those to be included. However, the standard does specifically state (in Sec. 8.2) that:

> *The symbolic names of the intrinsic functions are predefined to the processor. . . .*

A similar linkage technique may be utilized in connection with user-defined subprograms if transfer addresses provided in the object time data area are used. In essence, referring to a subprogram is similar to the task of referring to an undefined statement number and the same techniques of linkage apply.

In particular, since subprogram names are not deleted from the symbol table between subprograms, a transfer address location may be assigned at the first reference to the subprogram if it has not yet been compiled. If the subprogram is compiled before any other subprogram makes a reference to it, then the transfer address location must be reserved on recognition of the subprogram definition statement.

COMMON, DIMENSION and EQUIVALENCE

The layout of the object time data area is greatly influenced by the declarative statements COMMON, DIMENSION and EQUIVALENCE. In general, the FORTRAN programmer has no control over the absolute locations of data and variables, though by the use of COMMON and implicit dimensioning, certain items may be placed in memory in a prescribed order. In particular, a DIMENSION statement, while declaring arrays and requesting a certain number of locations, cannot force upon the compiler the manner in which the array is to be stored. As mentioned, the elements of the array may be stored in some random order and, provided that the object time routines could unwind this order to provide a certain requested element, the programmer would not be aware of the fact. Further, the definition of several arrays in a single DIMENSION statement does not imply that these arrays have any connection to each other. In fact, if the compiler does place these arrays in memory in that order, this is a peculiarity of the system, not of the language.

A COMMON statement, or a collection of COMMON statements, does imply contiguous placement of the variables and arrays. Blocked COMMON, on the other hand, does not imply contiguous blocks.

To complicate this vacuous concept, the programmer has the ability to equate variables by the use of the EQUIVALENCE statement. However, an EQUIVALENCE statement is doubly defined, or at least has two algorithms of organization depending on the data to be equated. Let us review some of the rules that control the use and meaning of the EQUIVALENCE statement.

1. Each pair of parentheses in the statement encloses the names of two or more variables that are to share the same memory location at object time.

2. Each quantity not mentioned in an EQUIVALENCE statement is assigned a unique location, except when that quantity appears in a COMMON statement.

3. Variables brought into the COMMON block by means of an EQUIVALENCE statement may increase the size of the COMMON block as originally specified in the COMMON statements. That is, if an array is brought into COMMON in such a fashion that some elements would fall outside the already established bounds of COMMON, then the size of the COMMON block must be increased to encompass the whole array.

4. Since the elements of an array are stored in consecutive locations from high-order address to low-order address, an array may not be brought into the COMMON block in such a way as to cause the array to extend beyond the upper bound of the block. In particular, as is a frequent occurrence, when the COMMON block is arranged to be the uppermost portion of memory, this protection ensures that some elements of the array do not lie in fictitious memory.

5. EQUIVALENCE may not rearrange COMMON. That is, two items already specified as existing in the COMMON block may not be equivalenced.

Certain effects implied in these rules must be observed. By rule 2, any variable not in COMMON and not specifically mentioned in an EQUIVALENCE statement must have a separate entity. Thus if the following statements appeared in a single subprogram,

```
DIMENSION A(10),B(3),I(2,3)
EQUIVALENCE (A(1),B(1))
```

memory would be arranged in the form:

```
A( 1) ≡ B(1)
A( 2) ≡ B(2)
A( 3) ≡ B(3)
A( 4)
A( 5)
A( 6)
A( 7)
A( 8)
A( 9)
A(10)
I(1,1)
I(2,1)
I(1,2)
I(2,2)
I(1,3)
I(2,3)
```

However, if the **EQUIVALENCE** statement were replaced by

```
EQUIVALENCE (A(10),B(1))
```

and then the arrays A and I were left in the same locations as previously, the array B would overlap the array I so that there would be an implicit **EQUIVALENCE**, such that I(1) and I(2) would share memory locations with B(2) and B(3), respectively. Thus a violation of rule 2 would exist. Hence the array I will have to be relocated.

```
A( 1)
A( 2)
A( 3)
A( 4)
A( 5)
A( 6)
A( 7)
A( 8)
A( 9)
A(10)     B(1)
          B(2)
          B(3)
I(1,1)
I(2,1)
I(1,2)
I(2,2)
I(1,3)
I(2,3)
```

Thus although there may be implicit equivalence of elements in those arrays that appear by name in the EQUIVALENCE statement, such that

```
EQUIVALENCE (A(5),B(2))
```

also implies

```
EQUIVALENCE (A(4),B(1)),(A(6),B(3))
```

undeclared implicit equivalences are invalid—except in COMMON. Thus given the statements

```
COMMON A(3), X,K
EQUIVALENCE (A(2),B(1))
DIMENSION B(3)
```

the COMMON block will be arranged as the sequence A(1),A(2),A(3),X,K and the following implicit equivalences will be valid

```
B(2) ≡ A(3)
B(3) ≡ X
```

Consider the statements:

```
COMMON A(3),X,K
DIMENSION B(5)
EQUIVALENCE (X,B(5))
```

Now after the compilation of the first statement, the COMMON block would be laid out as:

```
A(1)
A(2)
A(3)
X
K
```

Making the fifth element of B equivalent to X would imply the following equivalences, which are all valid since all variables are within the COMMON block: A(1) ≡ B(2), A(2) ≡ B(3), A(3) ≡ B(4), *but* element B(1) is now before (has a higher memory address) than A(1) in violation of rule 4.

By rule 5, the EQUIVALENCE statement may not cause a rearrangement of the variables in the COMMON area; that is, in effect, two items that already appear in COMMON may not be elements of an EQUIVALENCE statement group. This would seem to say that within one parenthetical group, there may appear only one item which is in COMMON. While this is true, the statement is not strong enough to prevent some implied equivalences, since the interaction of COMMON and EQUIVALENCE can cause a violation of violation of rule 5 that is not apparent in the statement, but that appears after certain groups in the EQUIVALENCE statement have been compiled. For example,

```
COMMON A(3), X, K
DIMENSION B(5)
EQUIVALENCE (X,B(1)),(B(2),A(3))
```

would require the following arrangement of COMMON:

```
A(1)
A(2)   X   B(1)
A(3)       B(2)
K          B(3)
           B(4)
           B(5)
```

which is illegal, as COMMON has been rearranged since the first definition of the area in the COMMON statement. To overcome this obstacle, it is stated (rule 3 implicitly) that a variable brought into COMMON must be regarded as having been declared as a COMMON variable after its first equivalencing. Thus group 2 of the above EQUIVALENCE statement is invalid.

While rule 3 states that a variable may be brought into COMMON in such a fashion that the length of the COMMON area is increased, there is one more rule, not stated above, which will prevent implicit overlap with variables not in COMMON (rule 2), that is:

6. Subprograms may not extend COMMON. Thus the COMMON block defined in a main program shall always be of a length greater than or equal to the COMMON blocks in subprograms. For this reason the location of the END OF COMMON must be communicated to all subprograms. For example, consider the following statements in a main line program and a subprogram:

```
PROGRAM MAIN        SUBROUTINE SUB
COMMON X,Y(3),K     COMMON A,B,C
                    DIMENSION X(2)
```

Such statements would implicitly equivalence the following variables:

Main Program	*Subprogram*
X	A
Y(1)	B
Y(2)	C

If the X array in the subprogram were placed immediately following the variable C in the subprogram, the following equivalences would be in effect:

Main Program	*Subprogram*
Y(3)	X(1)
K	X(2)

That is, the X array in the subprogram would effectively be in COMMON. Thus the END OF COMMON must be used to determine the object time addresses of non-COMMON variables in subprograms.

Conversely, if we allowed COMMON to be extended by a subprogram without recourse to extending COMMON in the main program, then some local variables in the main program would be implicitly in COMMON, hence violating rule 2. Rearranging COMMON as a result of an extension in a subprogram is not practical when subprograms may be compiled without reference to the main program. That is, where a subprogram may be compiled for use with several main programs, the compiler has no way to determine the size of the original array.

When the implementer is constrained to conform with the standard so that all the elements of the language are present (a reasonable request) and to permit all those features that have been overlooked (a questionable request), and thus is required to permit the extension of the COMMON area by a subprogram, he must follow one of two courses. He must either cause the examination of all COMMON block lengths to determine which is largest before assigning other memory space or ignore the rule stating that no implicit equivalences are permitted between variables that are outside of COMMON and the COMMON block, giving the programmer a diagnostic warning that this may cause some unusual results. In this instance, the insistence of not extending the COMMON area would seem to be a valid variance from the standard.

As a result of these intricacies, the "execution" of EQUIVALENCE statements at compile time is not straightforward and cannot necessarily be performed in the order set out by the programmer. For example,

```
COMMON A(3),X,K
DIMENSION B(5),C(2)
EQUIVALENCE (B(5),C(1)),(B(1),A(3))
```

would require a number of steps. Initially, that is, before the EQUIVALENCE statement is encountered, the COMMON area would be set out as:

```
A(1)
A(2)
A(3)
X
K
————————————END OF COMMON
B(1)
B(2)
B(3)
B(4)
B(5)
C(1)
C(2)
```

Then after the first EQUIVALENCE group has been executed, the following arrangement exists:

```
A(1)
A(2)
A(3)
X
K
————————————END OF COMMON
B(1)
B(2)
B(3)
B(4)
B(5)       C(1)
           C(2)
```

In practice, the only operation necessary to make this memory rearrangement would be to alter the addresses of the variables in the symbol table.

The next group within the EQUIVALENCE statement places B into the COMMON area, bringing with it the array C (adjusting the base addresses of both arrays), and shifts the END OF COMMON down by three words to encompass the whole set of variables and elements of arrays of which at least one element is now in COMMON. To remember that C is equivalent to a portion of the array B requires special handling. The final layout of memory will be:

```
A(1)
A(2)
A(3)    B(1)
X       B(2)
K       B(3)    ———————— Initial END OF COMMON

        B(4)
        B(5)    C(1)
                C(2)
———————————————————————— Final END OF COMMON
```

In processing COMMON, DIMENSION and EQUIVALENCE statements it is both uneconomical and frustrating to organize blindly the memory layout in the order of occurrence of the statements and variables within those statements. Consider Fig. 4.10. If all COMMON statements are collected primarily and processed, then a temporary END OF COMMON may be computed and permanent address assignments made to all variables that appear in that area since COMMON may not be rearranged. Next DIMENSION statements must be considered since the proper "execution" of the EQUIVALENCE statement depends on this knowledge. However, it is not known at this point which of the variables in the DIMENSION statement will eventually be brought into COMMON by equivalencing, and hence only the names and dimensions may be posted in the symbol table. To enable the later assignment of addresses to variables that do not appear in the COMMON area, a list of the variables occurring in the DIMENSION statement must be formed and reserved until the EQUIVALENCE statements have been processed. Having collected the dimensions of the arrays that may appear in the EQUIVALENCE statement, we must now collect EQUIVALENCE groups (that is, parenthesized groups declaring that the variables are to be stored in the same computer word) and scan these for the names of variables that have already appeared in the COMMON statement. Once these variables are recognized, all other variables in that group must be considered to be in COMMON and hence can be assigned permanent object time addresses on the basis of the original COMMON declaration and with reference to the preceding DIMENSION statement. Also these variables should be deleted from the list of variables not yet assigned addresses that was set up when the DIMENSION statements were scanned. Once such a group has been processed, *all* other groups must be scanned for variables that are now in the COMMON area. That is, if the first located COMMON variable in the first scan was found in the third and last group, the other two groups must be rescanned after the third group is processed in case any one of their variables is now in COMMON.

Once all groups in the EQUIVALENCE statement that contained either an explicit or implicit variable in COMMON have been eliminated, and the END OF COMMON has been adjusted, then all other groups must contain non-COMMON variables.

Symbols used in the Flowchart of the EQUIVALENCE Algorithm

EOC	The address of the last element in COMMON, that is, the address of the END OF COMMON.
DLIST	A list of variable names that are mentioned in the DIMENSION statements. This list is used after the execution of the EQUIVALENCE algorithm to assign unique addresses to these variables.
ELIST	A set of lists containing each group of the EQUIVALENCE statement.
ELIST(i)	The i-th sublist in ELIST.
NEXT	The next unused word in the available memory.
LLIMIT	The lower limit address in the algorithm.
N	The number of sublists in ELIST.
TADD	Temporary address.
SYMTAB	Symbol table.

Note: Since no element in an EQUIVALENCE group may be a formal parameter, all address assignments may be completed at compile time, and thus no object program is output from the algorithm for object time address computation.

FIGURE 4.10 Equivalence Algorithm
(Continued on next page.)

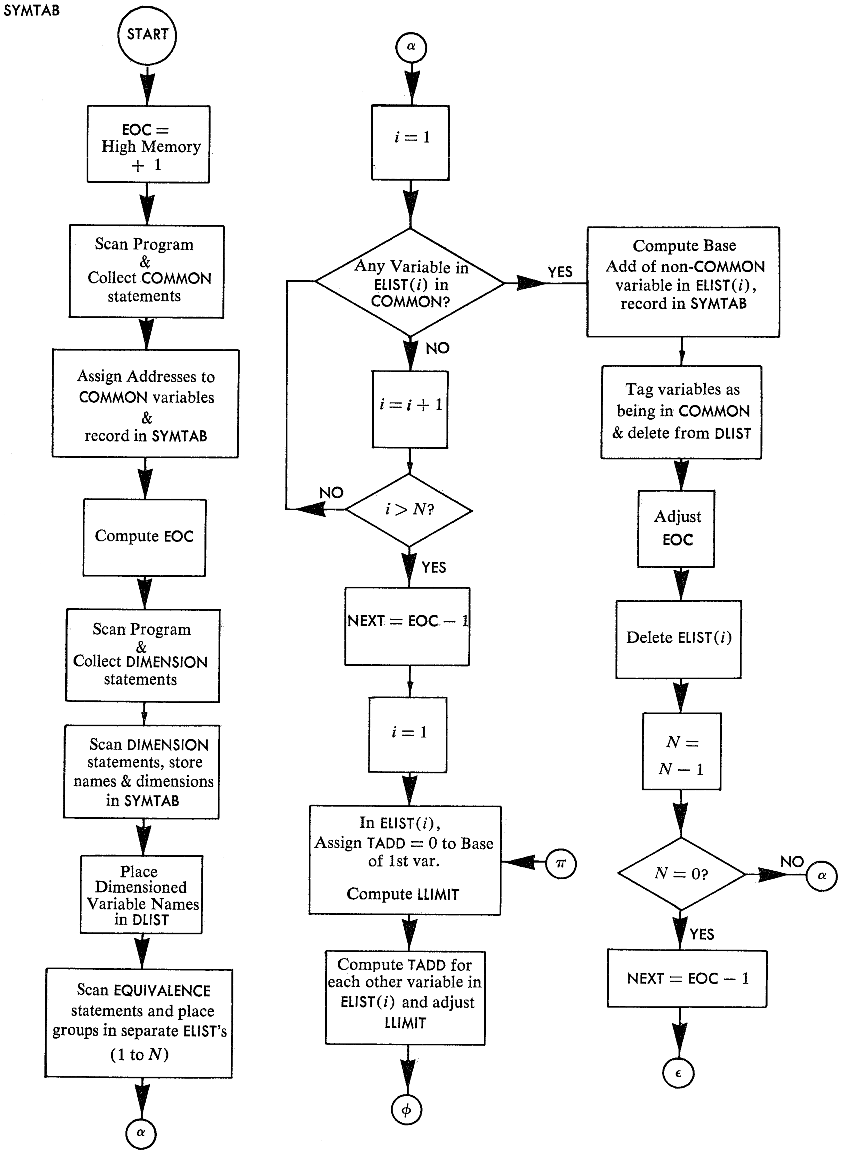

SYMTAB
START
EOC = High Memory + 1
Scan Program & Collect COMMON statements
Assign Addresses to COMMON variables & record in SYMTAB
Compute EOC
Scan Program & Collect DIMENSION statements
Scan DIMENSION statements, store names & dimensions in SYMTAB
Place Dimensioned Variable Names in DLIST
Scan EQUIVALENCE statements and place groups in separate ELIST's (1 to N)
α
α
$i = 1$
Any Variable in ELIST(i) in COMMON?
YES
NO
$i = i + 1$
$i > N$?
NO
YES
NEXT = EOC − 1
$i = 1$
In ELIST(i), Assign TADD = 0 to Base of 1st var. Compute LLIMIT
π
Compute TADD for each other variable in ELIST(i) and adjust LLIMIT
ϕ
Compute Base Add of non-COMMON variable in ELIST(i), record in SYMTAB
Tag variables as being in COMMON & delete from DLIST
Adjust EOC
Delete ELIST(i)
$N = N - 1$
$N = 0$?
NO
α
YES
NEXT = EOC − 1
ϵ

FIGURE 4.10 (continued)

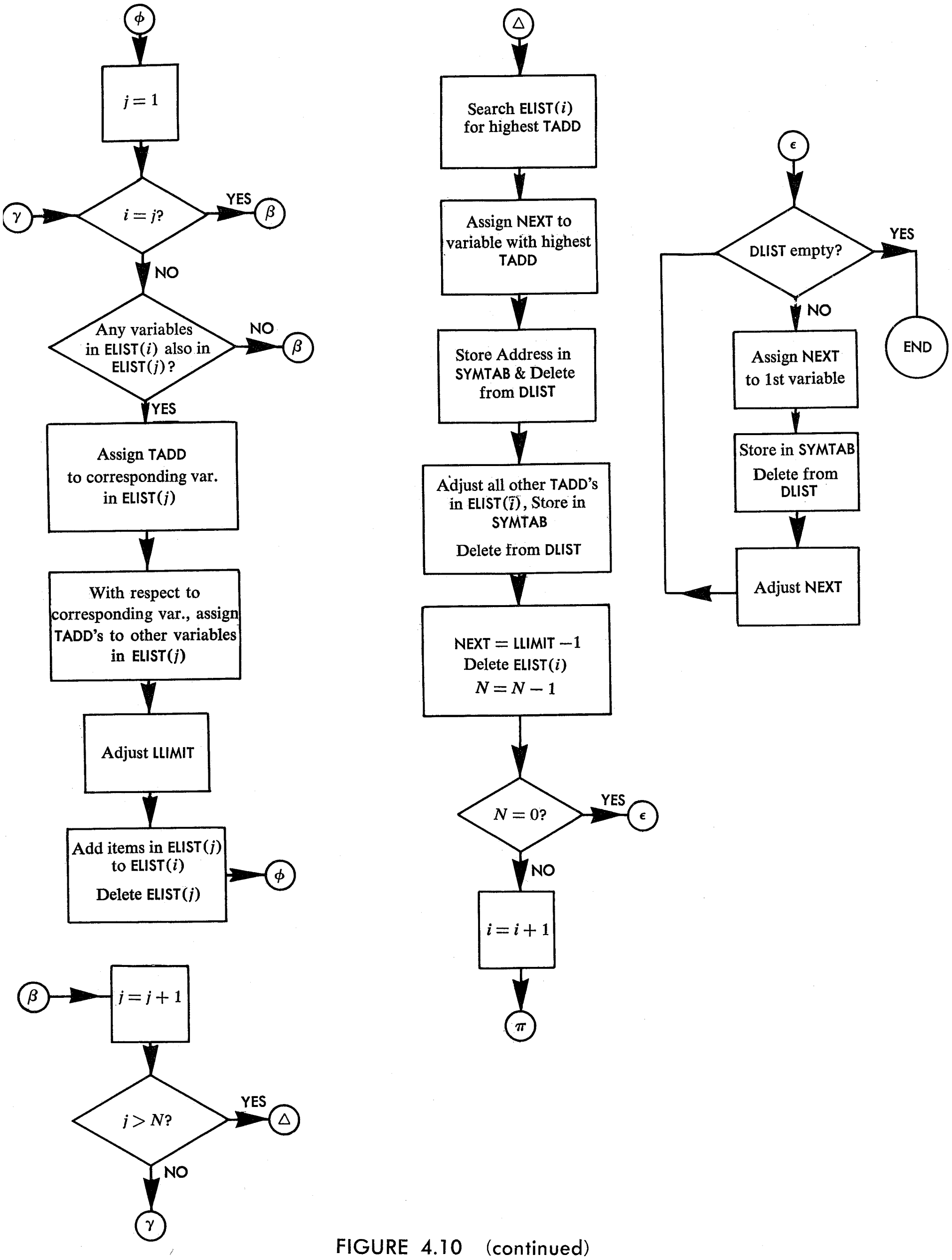

ϕ
j = 1
γ
i = j?
YES
β
NO
Any variables in ELIST(i) also in ELIST(j)?
NO
β
YES
Assign TADD to corresponding var. in ELIST(j)
With respect to corresponding var., assign TADD's to other variables in ELIST(j)
Adjust LLIMIT
Add items in ELIST(j) to ELIST(i)
Delete ELIST(j)
ϕ
β
j = j + 1
j > N?
YES
Δ
NO
γ
Δ
Search ELIST(i) for highest TADD
Assign NEXT to variable with highest TADD
Store Address in SYMTAB & Delete from DLIST
Adjust all other TADD's in ELIST(ī), Store in SYMTAB
Delete from DLIST
NEXT = LLIMIT −1
Delete ELIST(i)
N = N − 1
N = 0?
YES
ε
NO
i = i + 1
π
ε
DLIST empty?
YES
END
NO
Assign NEXT to 1st variable
Store in SYMTAB
Delete from DLIST
Adjust NEXT

FIGURE 4.10 (continued)

Since these groups have no fixed address on which to base their absolute address and much reshuffling may take place, for example, the first element in each group is arbitrarily assigned an address, each group must be scanned to determine its *spread.* The spread is the number of words that are affected by the arrays and variables in the group after the appropriate elements have been assigned to the same storage location. For example, if we have the statements

```
DIMENSION A(10),C(5)
EQUIVALENCE (A(8),C(2))
```

then regardless of the absolute address of each element, the group would be laid out as follows

```
A( 1)
A( 2)
A( 3)
A( 4)
A( 5)
A( 6)
A( 7)    C(1)
A( 8)    C(2)
A( 9)    C(3)
A(10)    C(4)
         C(5)
```

and the *spread* of the group would be 11 words. The uppermost element in this list or layout is A(1); thus if we were to place this group into memory from the top down, A(1) would be the element that would determine the placement of all other elements since it would have to fit into the next available location. Let us assign this key element the relative address of 000. The element C(1) would then have a relative address of −006, and the last element in the group (C(5)) would have the relative address of −010. Subsequent to these relative assignments, all other groups would have to be scanned to locate other occurrences of the same variables. Thus with the statements

```
DIMENSION A(10),C(5),X(3),Y(10)
EQUIVALENCE (A(8),C(2)),(X(1),Y(1)),(C(5),Y(3))
```

which contain the same dimensions of A and C as before with the same equivalence group, we would find that after relative addresses have been assigned to A and C, the third group also contains a reference to the array C.

Thus the Y array can be brought into relation with the elements of the first group and may be assigned relative addresses. Thus Y(1) will have a relative address of −008, and the lower limit of the group will be at −017.

The variables A,C and Y are now in a single grouping. A further scan shows that the only remaining group contains a reference to Y, and thus its associated variable (X) is also part of the grouping. Since the first element of the X array is equivalent to the first element in the Y array, their relative addresses must be the same; that is, the relative address of X(1) is −008. Further, since the X array does not extend the spread of the grouping, the lower limit relative address is still −017. The final arrangement of these arrays is

000	A(1)			
−001	A(2)			
−002	A(3)			
−003	A(4)			
−004	A(5)			
−005	A(6)			
−006	A(7)	C(1)		
−007	A(8)	C(2)		
−008	A(9)	C(3)	Y(1)	X(1)
−009	A(10)	C(4)	Y(2)	X(2)
−010		C(5)	Y(3)	X(3)
−011			Y(4)	
−012			Y(5)	
−013			Y(6)	
−014			Y(7)	
−015			Y(8)	
−016			Y(9)	
−017			Y(10)	

At this point, all groups in the EQUIVALENCE statement that were interconnected have been eliminated, and a list of variables whose permanent addresses may be assigned has been built up. Scanning the temporary relative addresses of the arrays in this list, we find that the highest address has been assigned to the first element of A. Thus this element can be placed in the next available absolute address, and all other elements and arrays can be given permanent addresses. The next available location for subsequent assignments may be computed from the lower limit relative address.

It may appear unnecessary to search for the largest temporary address since that first one assigned happened to be the largest; however, if the EQUIVALENCE statement had been written as

```
EQUIVALENCE (C(2),A(8)),(X(1),Y(1)),(C(5),Y(3))
```

then

C(1) would have been assigned the relative address of	000
A(1)	+006
Y(1)	−002
X(1)	−002

and the lower limit address would be −011. It would be possible to adjust all relative addresses so that the highest address is 000 at all stages of the process, but such repeated adjustment would be wasteful of time and can be easily omitted and replaced by the final search for the maximum relative address.

This procedure of tracking down the implicit groups in an EQUIVALENCE statement and assigning relative addresses can be repeated until all groups have been eliminated and, in effect, the EQUIVALENCE statement is empty.

Once all variables mentioned in the EQUIVALENCE statement have been assigned permanent addresses, the list of DIMENSIONed variables must be scanned to assign addresses to the remaining arrays. These may be arranged sequentially through the available memory. The process is shown in Fig. 4.10.

An Extension to the Equivalence Concept

USASI standard FORTRAN specifically states (Sec. 7.2.1.4) that the element in an EQUIVALENCE group may not be a parameter of a subprogram. This is mainly because of the difficulty in implementing a routine capable of handling all the cases that could arise if an EQUIVALENCE statement in a subprogram were able to contain a parameter wherever a simple variable is permitted by the standard. Further, the mention of the elements associated with or equivalenced to this parameter in other statements within the subprogram would require special handling. Let us consider the various situations in which a parameter might appear as an element of an EQUIVALENCE group in a subprogram.

Consider primarily the case in which a local simple variable in the subprogram is made equivalent to a single element that is a parameter of that subprogram. For example, consider the statements

```
PROGRAM MAIN
. . .
. . .
CALL SUB(X)
. . .
. . .
END
SUBROUTINE SUB(Y)
EQUIVALENCE (R,Y)
. . .
. . .
END
```

As will be discussed in Chapter 9, the problem of referring to an argument of a subprogram (in this case X) by way of the parameter list (in this case the single item Y) is easily accomplished by assigning an object time location to the parameter and placing the address of the argument into this location by using the interface (or linking) routine. All references to the parameter may then be made indirectly so as to pick up the address of the argument. Thus in this example of an EQUIVALENCE statement, the simple local variable may be assigned the same object time address as the parameter Y and may be tagged as a parameter so as to cause the compilation of indirect addresses in other statements. Thus no rules of the EQUIVALENCE statement would be violated other than that presently in question.

Consider the case where two parameters are placed in the same EQUIVALENCE group, for example,

```
PROGRAM MAIN
. . .
. . .
CALL SUB(X,Y)
. . .
. . .
END
SUBROUTINE SUB(A,B)
EQUIVALENCE (A,B)
. . .
. . .
END
```

This set of statements within the subprogram states in effect that, for all practical purposes, the two parameters are to be treated as a single variable and, in fact, if the two arguments were also equivalenced in the calling program, no problem would occur. However, in linking to the subprogram with two separate arguments where each argument has a distinct value, the question of which value is to be transmitted to the subprogram is unresolvable without some very special rulings. This case is similar to the rule that states that two elements that appear in COMMON may not be equivalenced, and therefore a small addendum to this rule would not seriously affect the meaning of the EQUIVALENCE statement.

When a local variable is equivalenced to an element in an array that is a parameter of the subprogram, a solution is not unattainable and in fact is quite practical. Consider the statements

```
PROGRAM MAIN
. . .
. . .
DIMENSION Y(50)
. . .
. . .
CALL SUB(Y)
. . .
. . .
END
SUBROUTINE SUB(Z)
. . .
. . .
DIMENSION Z(50)
EQUIVALENCE (Z(20),R)
. . .
. . .
END
```

In the subprogram a single data word is to be reserved to contain the address of the first element of the argument which corresponds to the parameter Z, and a special computation is required to locate each element in the array. Thus if the compiler sets a special tag in the symbol table against the name R, it is possible for all references to that variable to be compiled as a reference to the parameter Z with a computation to locate the actual placement of the value that is assigned to R or in which the result of some computation is to be stored. That is, whenever R appears in the subprogram, a set of instructions would be inserted into the object program which would locate the address of the parameter Z(20).

Finally, consider the case in which a variable that is a parameter is equivalenced to a local array in the subprogram:

```
PROGRAM MAIN
. . .
. . .
CALL SUB(M)
. . .
. . .
END
SUBROUTINE SUB(N)
DIMENSION L(10)
EQUIVALENCE (N,L(3))
. . .
. . .
END
```

At object time, the memory word which is reserved for the parameter N contains the address of the variable M and is tagged as a parameter so that all references to N are made indirectly so as to pick up the address of the argument. However, since L(3) is equivalent to N, any reference to an element in the L array has to be made on the basis of a special sequence of instructions which will locate the element needed. That is, a set of instructions which will compute the base of the array from the address contained in N must precede each reference to an element in the array.

However, if an element of the local array is equivalent to a single variable, then the elements that are not mentioned in the EQUIVALENCE statement must be implicitly equivalent to the words surrounding the base variable. In the above example, L(1) is implicitly equivalent to the second word above the word used to store M, and L(10) is equivalent to the seventh word below M. Thus there is a violation of rule 2. Similarly, in the previous case, if the array defined in the subprogram uses elements that are outside the range of the argument array, rule 2 is violated and spurious results may arise in any other statement that uses these elements.

In summary, EQUIVALENCE statements in subprograms that involve parameters of the subprogram are allowed provided that (a) no two (or more) parameters appear in the same parenthesis group or can in any way be implicitly equivalenced and (b) the local variables in a group are restricted to simple variables.

5

Control Statements

There are eight types of control statements:

GO TO statements,
arithmetic IF statements,
logical IF statements,
CALL statements,
RETURN statements,
CONTINUE statements,
DO statements and
program control statements.

This chapter will be limited to only six of these statement types; CALL and RETURN will be discussed in Chapter 9 in conjunction with considerations of subprogramming and the techniques of compilation within subprograms. Program control statements do not, in fact, contribute significantly to the algorithm described by the program and are, in general, compiler or machine (as opposed to language) oriented in their implementation. In particular, the STOP statement in either form:

$$<STOP\ statement> := \mathsf{STOP}\ \{<digit>\}_1^5 | \mathsf{STOP}$$

causes the termination of the program at execution time. However, the manner of termination depends on the specifications of the monitoring system, if one is present. In an unmonitored system, such as used on smaller computers in an open shop, it is satisfactory to allow the program to halt the computer and to expect the operator to take over control. In a supervised operation, control should be returned to the monitor which will clean up the work areas and provide the system and programmer with information determined by the designers. Thus in the latter case, the computer will not in fact STOP but rather the execution will be terminated.

On the other hand, the execution of a PAUSE statement does actually cause the computer to hesitate, though the degree of hesitation is a function of the system. For example, in both an unsupervised and supervised system the computer may cease operation and wait for operator intervention, whereas in a time-shared operation the execution of the program may cease while another program is executed and only after the appropriate information has been received by the system will the program be placed in the queue for further execution.

Similarly, in both program control statements, the display of the string of digits, if present, is determined by the compiler writer or specifications of the monitor system. For example, in some systems the string of digits is displayed in a light display, whereas in others the string is output through a printer or typewriter.

Statement Identifiers

Since the compilation of control statements is heavily dependent on the handling of statement identifiers in the symbol table routines, let us review the situations which may occur in recording statement number references in a one-pass system.

Case 1. Reference is made to a statement number that has already been encountered in the program. In this case the symbol table must contain: (a) a statement number tag, (b) a representation of the number, (c) the object time address of the first instruction (or first word) of the referenced statement, and (d) a tag to indicate that (c) is defined.

Case 2. Reference is made to a statement number not yet encountered. Then a symbol table will be created, containing (a) a statement number tag, (b) the representation of the number, (c) the object time address of a data word that will eventually contain the object time address of the first word of the referenced statement, and (d) a tag to indicate that (c) is an indirect address.

At the time that the statement number is encountered, tag (d) should be investigated, and if the tag indicates that (c) contains a direct address, then the statement number is doubly defined and an error message should be emitted. If the statement number is not found in the table, then the data for case 1 should be generated. When case 2 has already occurred, the object time address of the statement must be provided to the loader routine for placement into the reserved data word. At the same time, address (c) should be updated to the actual instruction address and tag (d) set accordingly.

GO TO Statements

Let us now consider the compilation of the unconditional or assigned GO TO statements. That is, the statements

$$<\textit{GO TO statement}> := \text{GO TO } \{<\textit{statement number}>| \\ <\textit{integer variable}>\}^1$$

In the first case above, in a one-pass compiler, the form of the compiled instruction depends on the previous encounter of the statement number. In case 1, the instruction may be compiled as

```
† B      Address(n)
```

where Address(n) is the actual object time address of the statement labeled n.

In case 2, where the reference is forward, compile

```
B      -Data(n)
```

where Data(n) is the address of the object time data word that contains the actual address of the statement labeled n, and the minus sign preceding the address portion of the symbolic instruction indicates indirect addressing.

Alternatively, in case 2 the compiled instruction in a one-pass system may be left blank (or filled with any arbitrary address) and then be overlaid

† B is the mnemonic for BRANCH or unconditional JUMP.

with the actual address by the loader routine. To accomplish this, it would be necessary to append a list of addresses of incomplete instructions to the preliminary entry in the symbol table containing the data on the statement number. The acceptability of such a procedure depends on considerations of available memory at compile time, a specialized loader and extra loading time, or simply the extra cost of implementation.

In the case of the assigned GO TO statement, the instruction must always be compiled as an indirect branch, since the address can never be determined at compile time; hence GO TO α compiles to

```
B        −Data(α)
```

where Data(α) is the object time address of the variable α.

While considering the assigned GO TO statement, let us consider the ASSIGN statement. The general form is

<ASSIGN statement> :=
ASSIGN<*statement number*>TO<*simple integer variable*>

In case 1, where the referenced statement has already been encountered, we may compile

```
† ENA     Address(n)
  ST      Address(i)
```

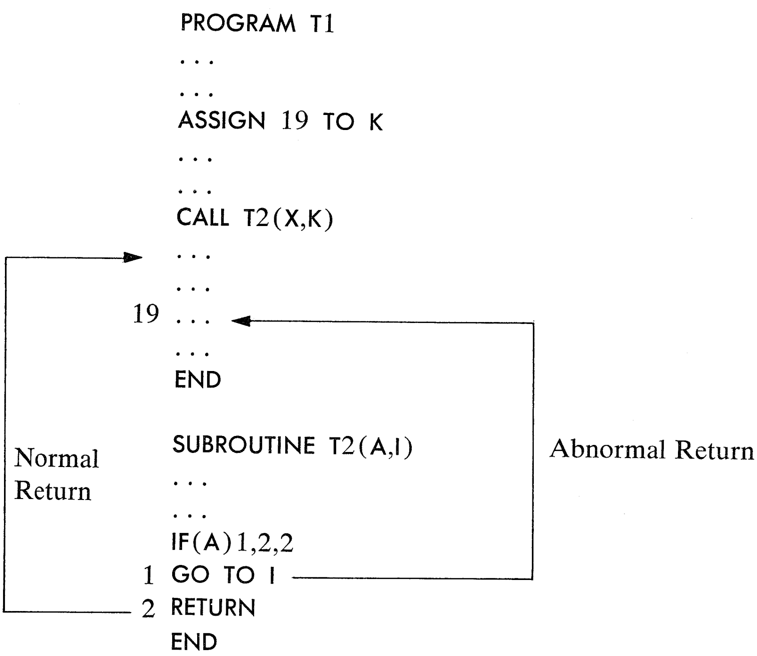
where Address(n) is the object time address of the first word in the referenced statement and Address(i) is the address of the variable i. In case 2, we must use a data table reserved word, which will be filled with the address of the referenced statement at load time, to compile:

```
† LDA     Data(n)
  ST      Address(i)
```

An interesting situation may occur in systems that permit the assigned GO TO statement and have subprogram capabilities. If a statement number has been assigned to an integer variable, then that variable may be used as an argument in a subprogram CALL. Hence, within the subprogram an assigned GO TO statement that uses that variable can cause control to be returned to the calling program without a RETURN statement or necessarily executing the statement immediately following the CALL statement.

† LDA is the mnemonic for LOAD ACCUMULATOR with the data at the address. ST is the mnemonic for STORE the data in the accumulator into the address in the instruction. ENA is the mnemonic for ENTER ACCUMULATOR with the data in the address portion of the instruction.

For example, consider the following job:



```
      PROGRAM T1
      . . .
      . . .
      ASSIGN 19 TO K
      . . .
      . . .
      CALL T2(X,K)
      . . .
      . . .
   19 . . .
      . . .
      END

      SUBROUTINE T2(A,I)
      . . .
      . . .
      IF(A)1,2,2
    1 GO TO I
    2 RETURN
      END
```

While one may argue that this is an illegal use of an assigned GO TO statement and a CALL statement, there is no specific rule prohibiting the use of a formal parameter in the assigned GO TO statement.

As will be discussed later, formal parameters have no private existence since they take values from the CALL arguments. Thus, the variable I will contain the address of the argument K (in this case), and K will contain the address of the statement labeled 19. Now in the calling program we may compile

```
B     −K
```

for the FORTRAN instruction GO TO K, but to compile

```
B     −I
```

in the subprogram would lead to an error since I contains not −K but the address of K itself. Thus if the variable in an assigned GO TO statement is a formal parameter, we must compile the set of instructions

LDA −I Contents of −I to ACC, i.e., contents of K to ACC, i.e., address of 19 to ACC.

[†]ST *+1 Store contents of ACC in next instruction; i.e., put address of statement numbered 19 in the next instruction.

B *−* Branch to address to be filled in.

With this scheme, the statement number address may be transmitted through many levels of subprogram provided that all references to formal parameters in a subprogram are addressed indirectly. Thus, the following program would execute correctly with cascading indirect addressing:

```
      PROGRAM T1
      . . .
      . . .
      ASSIGN 19 TO K
      . . .
      CALL T2(A,K)
      . . .
      . . .
   19 . . .
      . . .
      END

      SUBROUTINE T2(X,I)
      . . .
      CALL T3(X,I)
      . . .
      RETURN
      END

      SUBROUTINE T3(Y,J)
      . . .
      . . .
      CALL T4(Y,J)
      . . .
      RETURN
      END
```

[†] The symbol * stands for the address of *this* instruction.

```
SUBROUTINE T4(Z,K)
. . .
. . .
GO TO K
. . .
RETURN
END
```

The statement CALL T2 in the main program will place the address of K into the space reserved for the formal parameter I. CALL T3 in subroutine T2 will transfer −I to J since all references to formal parameters in a subprogram are referenced indirectly. CALL T4 will transfer −J to K. Then the compiled instruction LDA −K in subprogram T4 will chain through three levels of indirect addressing to pick up the address of the variable K in the main program and hence place the address of statement number 19 in the accumulator. This address will then be stored in the branch instruction.

In computers without cascading indirect addressing, only one level of assigned GO TO may be permitted in a subprogram, and with this restriction, *all* assigned GO TO statements using formal parameters should be rejected.

Let us now step aside from these machinations on the assigned GO TO and consider the computed GO TO, which takes the form:

<computed GO TO statement> := GO TO(*<statement number>* {,*<statement number>*}$_0^\infty$),*<integer variable>*

In its standard form, this may be compiled to the instructions:

```
ENA  *+3    Address of branch to ACC
ADD  I      Add contents of address I
ST   *+1    Store in branch instruction
B    −0     Branch indirectly
```

This set is to be followed by a list of addresses of the statement numbers, modified to indirect addresses, if necessary, for forward references.

Such a compilation does not protect the user from attempting to use a value of the index that is zero, negative or greater than the number of items in the parenthesized list. This would be an object time error, and thus any instructions to check this value must be included in the target language. However, the rules of the language do not provide any directions as to the course of action to be taken if the index is greater than the number of items. In fact, the USASI specification defines the statement as follows:

> 7.1.2.1.3 *Computed* GO TO *statement.* A computed GO TO statement is of the form:
>
> $$\text{GO TO(}\ k_1, k_2, \ldots, k_n),i$$
>
> where the k's are statement labels and i is an integer variable reference. . . . Execution of this statement causes the statement identified by the statement label k_j to be executed next, where j is the value of i at the time of execution. This statement is defined only for values such that $1 \leqslant j \leqslant n$.

Irrespective of the bland statement that this statement is defined only within the appropriate range, most compilers, or, in fact, the object time routines, provide for the contingency of a value of the index outside the range. For example, the FORTRAN 3600 specifies that for the purpose of execution, a value of the index greater than the number of items in the list is taken to have a value equal to the limit. That is, if $j > n$, the next statement to be executed will be that labeled k_n. On the other hand, KINGSTON FORTRAN II uses modular arithmetic to pick out the appropriate statement number based on the limit. That is, j is taken to have the value $j - n(int(j/n))$ where *int* is the integer function of the argument.

Such routines take considerable storage and cannot be stored (or compiled) efficiently in line when several computed GO TO statements are anticipated in a program. Using a special routine within the system, we may compile the following:

```
ENA     *+2
B       COMGO
        Address(i)
        n
        Address(k1)
        Address(k2)
        . . .
        . . .
        Address(kn)
```

where COMGO is the entry address to a routine that checks the value of the index against the limit (provided by the compiler in the appended list), corrects the value if necessary and then picks out the correct statement number address from the list. Since the accumulator contains the address of the list on entry to this routine, the list may be accessed from the routine easily. Similarly, knowing both the location and the size of the list, the compiler will have no problems in providing the correct starting location for the next set of instructions.

Before continuing with the discussion of control statements, let us consider the implementation of some nonstandard features which do not cause the compiler writer any major problems and do not detract from the available space at object time or speed of execution. For example, if one is prepared to implement the assigned GO TO statement using a simple integer variable, then there is little difficulty in extending the power of this statement to include subscripted variables. As will be shown later, the compilation of a subscripted variable reference is merely the use of the algorithm to generate the address of the variable. Thus if reference is made to the arithmetic scanning routine to pick out the address of the variable instead of simply referring to the symbol table routine, statements such as

```
ASSIGN 103 TO L(3*I+5)
. . .
. . .
GO TO L(J)
```

are compilable and executable. However, in a single accumulator machine, care must be taken to ensure that the computations to determine the address of the element in the array do not conflict with the load and transfer instructions necessary to place the address of the statement number in the branch instructions. Thus the simple pair of symbolic instructions for the compilation of the ASSIGN statement must be preceded by those necessary for the determination of the element address. The above ASSIGN statement might be compiled to:

```
ENA   =3              Enter literal 3 into the ACC.
MUL   I               ACC*I to ACC.
INA   =5              Increment the ACC by literal 5.
RVSG                  Reverse the sign of the ACC.
INA   Address(L(0))   ACC+base of array L to ACC.
ST    *+2
ENA   Address(103)    Address of first word of reference
                      statement to ACC.
ST    *-*             Store ACC in location to be provided.
```

If the arithmetic expression routine provides information regarding its actions to the calling routine, then the calling routine can compile in one of two manners. If the arithmetic expression routine provides the current address of the result of the compilation as it would appear at object time, then the calling routine is able to determine whether in line instructions have been compiled. For example, if the second element of the ASSIGN statement is a simple variable, the arithmetic expression routine has done no work past that of determining the address of that variable; thus the calling routine will report that the address of the result is that address. On the other hand, if the arithmetic expression routine is forced to compile in line instructions the address of the result is the accumulator. On the basis of these reports, the computer will compile

```
ST    *+2
ENA   Address(n)
ST    *-*
```

for the case when the called routine reports that the result is in the accumulator, or

```
ENA  Address(n)
ST   i
```

when the result is not in the accumulator.

Similarly, if variables are allowed in a GO TO statement, then it is feasible to use the same process in connection with the computed GO TO statement such that the following statement would be valid:

```
GO TO(I, 1, 17, J(K),99),L
```

where both I and J(K) have appeared on the right-hand side of a previously executed ASSIGN statement. Once again, however, the order of processing in the compiler must be revised. For example, when the list contains only statement numbers and simple variables, the list appended to the instructions that link to the COMGO routine will contain either the actual address of the statement referenced provided that the statement has already been encountered or an indirect address to a forward reference or to a variable reference. When subscripted variables are included in the list, the one for one ordering of the object time list will be disturbed by the presence of instructions to compute the actual location of that element. Thus we may propose to perform all these subscripting instructions before entering the COMGO routine, placing the appropriate addresses into the appended list. However, this technique would mean that prior to each and every execution of the computed GO TO all subscripting computations would be executed *whether or not that reference were used.* This could seriously slow down the execution of this type of statement.

Alternatively, since it may be recognized that this computed GO TO statement causes a complete break in the sequence of execution of the program, it would not affect the operation materially if an extra routine were slipped in between a branch from the GO TO statement routine and a branch to the final destination. Thus if the appended list contains the addresses of the first instructions in routines to compute the address of elements in arrays, instead of addresses of the next instruction in the program (direct or indirect), the computation of the location of the element in the named array need only be performed when necessary. Consider the GO TO statement:

```
GO TO (13, K, 7, J(I), M(N)), L
```

which would be compiled to the sequence:

```
      ENA   *+2                        Address of list to ACC
      B     COMGO                      Jump to COMGO routine
            Address(L)
            =5
            Address(13)
            -Address(K)
            Address(7)
            Address(LINK1)             Address of subscripting routine
            Address(LINK2)             Address of subscripting routine
LINK1 LDA   I                          Compute address of J(I)
      RVSG
      INA   Address(Base of J)
      ST    *+1                        Store address
      B     -0                         Branch indirectly
LINK2 LDA   N                          Compute address of M(N)
      RVSG
      INA   Address(Base of M)
      ST    *+1                        Store address
      B     -0                         Branch indirectly
```

Since the index of a computed GO TO merely provides a source of data, this extension can be carried further with any integer expression in place of the index such as:

```
GO TO (I, 1, 17, J(K), 99), I1+(J3**2)/3
```

This expression may be computed in the place of an index before entry to the COMGO routine, and the location of the result may be placed into the appropriate word location in the appended list. That is, the result of this calculation must be placed into temporary storage if arithmetic operations are involved in the computation; if the index is merely a subscripted variable with a variable subscript, the address of that element may be placed directly into the list. Thus the index

I+J*3

would compile to the instructions

```
LDA  J              J to ACC
MUL  Address (3)    ACC*3 to ACC
ADD  I              ACC+I to ACC
ST   TEMP           ACC to temporary location
```

The appended list would then contain the address of the temporary location provided by the compiler. On the other hand, if the index is a subscripted variable with a variable subscript, the subscripting computations must be performed before entry to COMGO and the address of the element put into the list at object time. That is, an index of I(J) would cause the compilation of the following instructions

```
LDA   J                     J to ACC
RVSG                        Reverse sign of ACC
ADD   Address(Base of I)
ST    *+3                   Store address in list
```

Indices that are simple variables or subscripted variables with constant subscripts may be compiled with no extra instructions and with the actual address of the index in the appended list.

Irrespective of these extensions to the computed GO TO statement, the overall power of the statement is somewhat diminished when the extension to the assigned GO TO statement to include subscripted variables is permitted.

IF Statements

FORTRAN IF statements may take one of four possible forms:

<IF statement> := IF(*<arithmetic expression>*)*<statement number>*
{,*<statement number>*}$_2^2$|IF(*<logical expression>*)
{*<statement number>*{,*<statement number>*}$_0^1$|
<FORTRAN STATEMENT body>}^{1}

The compilation of the interior of the parentheses of the IF statement is identical to that of the right-hand side of an arithmetic assignment statement or a logical assignment statement, respectively, and will be discussed in Chapter 8. In fact, the same routine that will compile these expressions may be used to compile the interior of the IF statement. In either case, however, the IF generator need only be aware of the type of interior expression (arithmetic or logical) and take into account that the result in either case will be left in the accumulator.

In the standard IF statements, the difference between those statements that reference statement numbers and those that contain the actual statement to be executed if the result of the expression is true, may be recognized easily by examination of the first character following the closing parenthesis. Only when an actual FORTRAN statement body is defined is the next character alphabetic. In fact, an examination of the possible statement bodies in the FORTRAN language reveals that the next character must be alphabetic in this case.

Once the latter type of IF statement has been recognized, there are several distinct steps to be undertaken:

1. On the basis that the result of the execution of the logical expression is residing in the accumulator and that a zero value indicates falsehood and unity represents truth, compile the single instruction

```
BZ      *—*
```

where BZ is the mnemonic for **BRANCH ON ZERO ACCUMULATOR**. The address of this instruction is left blank since the location of the next instruction to be executed is not known at this juncture. This instruction when executed at object time will cause control to branch around the set of instructions that will be compiled from the FORTRAN body when the result of the execution of the logical expression is false. Also, the address of the above instruction is stored for future reference.

2. Compile the FORTRAN body as any other FORTRAN statement by using the SIEVE and the appropriate generator. Note that the standard forbids the use of another IF statement (among others) as a statement body since this would be tantamount to permitting recursion in the language. If this were allowed, the address of the BZ instruction generated in step 1 would have to be placed in a push down list. In any case, the need for a

subsequent IF statement is made unnecessary by the use of the logical operators .OR. and .AND..

3. After compiling the statement body, insert the address of the next instruction, which is now known, into the BZ instruction by referencing the address that was stored by step 1. If the compiler is a load and go system, this insertion is simple; but when the object code is transmitted to some external device, the loader must overlay this address correctly, and it must be assured that the instruction is loaded before the address portion is overlaid. In a multipass system, the address portion of the BZ instruction may be filled in with a symbolic address and that address be used to label the instruction of the next statement.

In contrast to compilation of the GO TO where it is proposed that an indirect branch to a data word be used when the statement number has not been encountered, *backtracking* is more feasible in the case of an IF statement since it is known that the only instruction to be overlaid is the BZ instruction. Further, this process of backtracking to fill in the address is within the same generator, whereas in the case of the GO TO, backtracking must be accomplished by the routine that recognizes statement numbers in the label position of the FORTRAN instruction.

The process of compiling the branches to other types of IF statement may be combined into a single scanner and checker. Given a three element list such as a_1, a_2 and a_3, the following algorithm will suffice:

1. Extract the first number and store its address (from SYMTAB) in the word a_1.

2. If the delimiting character following the extraction of the first number is the end of the statement, then the IF statement is of the type

IF(<*logical expression*>)<*statement number*>

and may be compiled as simply

BNZ (a_1)

where BNZ is the mnemonic for BRANCH ON ACCUMULATOR NONZERO, and (a_1) is the address of the referenced statement. Following this compilation, the IF generator may be exited.

3. Extract the second number and store in a_2.

4. If the delimiter to the second number is the end of the statement, the class of IF statement is that of

IF(<*logical expression*>)<*statement number*>,<*statement number*>

where the next statement to be executed if the result of the execution of the logical expression is true is that referred to first in the list of exits and if false, the next statement is that labeled lastly. With the data stored in a_1 and a_2 this IF statement may be compiled in one of two manners, either

	BZ	(a_2)	If false, branch to second statement reference.
	B	(a_1)	If true, jump to first reference.
or	BNZ	(a_1)	If true, branch to first statement number.
	B	(a_2)	If false, go to second reference.

5. Extract the third statement number and store in a_3.

6. By default, the IF statement must be of the class of arithmetic IF statements, that is,

IF<*arithmetic statement*>)<*statement number*>{,<*statement* number>}$^{2}_{2}$

where the statement numbers refer to the exits corresponding to the results which are negative, zero and positive, respectively. By using the data stored in the elements a_1, a_2 and a_3, the following instructions may be compiled:

BZ	(a_2)	Branch on zero accumulator to address contained in a_2.
BN	(a_1)	Branch if accumulator is negative to address contained in a_1.
B	(a_3)	Branch to address contained in a_3. As a result of the previous branches this instruction is effectively a branch on positive accumulator.

Note that since the statement numbers were originally contained in the *a* elements, these elements may be replaced before compilation by the actual or indirect addresses of the referenced statements. Thus in the above instructions the elements have been enclosed in parentheses to show that the addresses are to be obtained from the elements *a* at compile time.

The seemingly peculiar disarrangement of these branch statements is chosen to overcome the peculiarity of some computers that give a sign to zero values. This occurs because in some machines the sign of the original accumulator value is retained when the result of a single operation is zero. Similarly, the second instruction is for a negative condition since, in some computers, the speed of execution of a test for a negative value is faster than that of the other tests. The order of the tests may, therefore, be machine dependent though in theory there is no particular significance to the order described above.

Let us now discuss the possibility of allowing *statement identifier variables* in IF statements. In standard IF statements, the distinction between those statements containing statement numbers and those containing a statement body after the closing parenthesis may be made by examining the first character following the parenthesis. However, if statement identifier variables are allowed, this test is not valid. Compared to variables used as a data source, such as in the index to a computed GO TO statement, which can easily be replaced by a complete expression, a statement identifier variable is merely the source of an address to which a reference may be made and, therefore, cannot be replaced or influenced by an expression. Thus the statement identifier variable is similar to the variable that occurs on the left-hand side of an arithmetic assignment statement. If the routine that scans arithmetic statements were divided into two parts, these would be:

1. The right-hand side scanner, which compiles instructions to execute the specified operations and leaves the result (a value) in the accumulator.

2. The left-hand side scanner, which compiles subscripting, if necessary, and provides the address of the variable. In an arithmetic assignment statement, it is into this address that the result of executing the right-hand side is to be placed.

Hence, the left-hand side scanner will check for the presence of any operator and "complain" if necessary, and from the result of the scanner, one may determine the difference between the two main types of IF statement. In particular, the scanner will probably believe that a statement number is an integer constant, which is not a valid "loadable," thus complain. However, if the manner of complaining is not the output of an actual error message, this task being left to the calling generator, then this special routine will enable the IF generator to scan across to the first delimiter. If this delimiter is a comma or the end of the statement, then the IF statement is the type that contains statement numbers (in the standard form), whereas if the delimiter is not one of this type, further scanning is necessary. This task may be taken over by the SIEVE routine and the statement handled as a normal logical IF statement with an embedded statement body. Some statement bodies, unfortunately, may be taken for variables, and thus exact specifications must be laid down with regard to the inclusion of special delimiting characters. For example, the exclusion of the spaces between the elements of the standard GO TO statement, such as

```
GOTO100
```

could cause confusion. However, if we insist that the blanks be included, then the left-hand scanner will light upon the blank as the delimiter. Similarly, an input statement without a blank between the keyword and the statement number or logical unit specifications could be scanned in either of two ways: the input statement could be taken for either a list of simple variables or a subscripted variable. For example, consider the scanning of the following statements:

```
IF(PASS.EQ.1)READ1,A
```

and

```
IF(IT.EQ.THAT)READ(12,61)A
```

If the embedded blanks are not included as part of the language, then the only alternative to obtain a correct scan would be to reserve certain variable character combinations such as those beginning with the characters READ, WRITE, GOTO, etc. However, the exponents of FORTRAN have always been able to claim that the language contains no reserved words. Further, if the extension of the language is such that previously valid programs will no longer compile or execute correctly, then the extension itself must be considered invalid.

When statement identifier variables are also subscripted, the order of executing the subscripting instructions will substantially affect the efficiency of the object program. That is, as with the extended computed GO TO, it is inefficient to perform the subscripting instructions when an identifier is not to be used as the exit to the statement. Thus any compiled instructions for subscripting that the left-hand scanner generates must be saved in the computer memory before being added to the object code. That is, if the subscripting instructions are placed in line as the scanner generates them, the following arrangement might occur in the worst case (the arithmetic IF with subscripted statement identifier variables):

```
. . .  )
. . .  }      Evaluate parenthesized expression.
. . .  )
ST   TEMP     Store result in temporary location.
. . .  )
. . .  }      Evaluate subscript of n2.
. . .  )
LDA  TEMP
BZ   Address(n2)
. . .  )
. . .  }      Evaluate subscript of n1.
. . .  )
LDA  TEMP
BN   Address(n1)
. . .  )
. . .  }      Evaluate subscript of n3.
. . .  )
B    Address(n3)
```

To alleviate this excessive amount of coding as well as the execution of unnecessary instructions at object time, the subscript calculations should be executed after the test of the result of the parenthesized expression:

```
     . . .  )
     . . .  }      Evaluate parenthesized expression.
     . . .  )
     BZ   S2       Branch to subscript routine no. 2.
     BN   S1       To routine no. 1.
     B    S3       To routine for positive case.
S1   . . .  )
     . . .  }      Evaluate subscript of n1.
     . . .  )
     B    Address(n1)
S2   . . .  )
     . . .  }      Evaluate subscript of n2.
     . . .  )
     B    Address(n2)
S3   . . .  )
     . . .  }      Evaluate subscript of n3.
     . . .  )
     B    Address(n3)
```

Obviously, if the statement identifier variable is unsubscripted, this extra set of instructions is not necessary, and the exit may be made directly from the first set of tests.

The DO Statement

The FORTRAN DO statement, in its standard form, is a combination of statements. In fact, the general statement

```
DO 1 I=J,K,L
```

can be implemented equally as well by the statements:

```
     I=J
   2 ...
     ...
     ...
   1 I=I+L
     IF(I.GT.K)11,2
  11 ...
     ...
```

where the statement identifiers 2 and 11 are provided by the compiler and do not, in fact, conflict with other statement numbers in the program. In symbolic code, the above DO statement could be compiled as:

```
        LDA  J
        ST   I
NEXT    ...
        ...
        ...
ST1     LDA  I
        ADD  L
        ST   I
        SUB  K
        BN   NEXT
        BZ   NEXT
        ...
        ...
```

When the parameters of the DO list are integer constants, advantage should be taken of any machine instructions that use their address portion as the actual data to be used in the instruction. That is, if the machine has instructions such as INA (INCREMENT ACCUMULATOR) or ENA (ENTER ACCUMULATOR) where the increment or value to be used is actually within the instruction itself, then data table storage space may be saved by using these instructions where possible. Thus the FORTRAN statement,

```
DO 13 N=1,K,3
```

may be compiled to:

```
        ENA  =1      Literal 1 to ACC
        ST   N
NEXT    . . .
        . . .
        . . .
ST13    LDA  N       N to ACC
        INA  =3      Increment ACC by literal 3
        ST   N
        SUB  K
        BN   NEXT
        BZ   NEXT
```

When a DO statement occurs in a subprogram and any one of the parameters or the index itself is a formal parameter, indirect addressing references to the address of that formal parameter will take care of the necessary chaining (or cascading) through to the actual value of the element.

The main drawback in implementing the DO statement is the necessity to split the object coding between the beginning and the end of the range. One technique for obtaining this split is to add a set of special information to statement number entries in the symbol table.

When a DO statement is encountered, the statement number defining the last statement within the range is not yet defined. However, the statement number may already be in the symbol table as it may have been referenced in an exterior DO range. Since the address is not yet defined and cannot be referenced by any other control statement from outside the range, a special tag may be included in the symbol table entry which will define this statement number as the termination of a DO range. Emanating from this symbol table entry will be a linked list containing five entries per sublist:

NEXT, LIMIT, INCREMENT, INDEX and LINK

where LIMIT, INCREMENT and INDEX are the addresses of the parameters in the DO statement; NEXT is the address of the first instruction in the DO range after the initialization instructions; and LINK is the address of the next sublist (which will be undefined if this is the last sublist). That is, this is the data pertinent to the innermost DO range that uses this statement number up to this point in the compilation. This list and its sublists are constructed as the DO statements are encountered and are destroyed as the range terminating statements are recognized. The last sublist emanating from the symbol table entry refers to the innermost DO range for which object time instructions should be generated.

As originally conceived a DO statement was based on the control of the range by means of index registers where each register consisted of three parts: index, limit, increment. In such a machine, the incrementation and testing of the index could be performed in a single instruction, and when a new index value was less than the limit, this instruction would also transfer control to the instruction defined by the address in the instruction address field. Otherwise, the normal sequence of instructions would be followed. Only one instruction was split off to reside at the end of the range, and the instruction to initialize the register (all parts) occurred at the location in the object code equivalent to the location of the DO statement in the source program.

This mode of implementation created several restrictions:

1. The depth of DO nests could not exceed the number of available index registers.

2. The index of the DO loop, *i,* for example, was not related to the variable *i* in the rest of the program, but was related to all the references to *i* within the range.

3. Because of the limited availability of index registers, the same register had to be used for several unrelated loops. Hence, since the compiler could not ascertain the logical flow of the program, a DO index had to be considered undefined outside the range.

4. The index could be incremented only.

5. The range had to be traversed at least once, even if the original (initial) value of the index was greater than the limit.

6. Real variables or constants were not permitted as elements of the DO statement.

7. For obscure reasons, expressions were not permitted as elements of the list.

8. The value of any element in the DO list could not be altered within the range since their values were stored in inaddressable registers.

Let us now consider these restrictions in the light of the type of implementation described in this chapter.

1. Since index registers are not utilized, the depth of a DO nest is limited only by the available space at compile time for storing the list of uncompleted DO statement elements.

2. Since, in the original implementation, the index was unrelated to a similarly named variable in the rest of the program and was, in fact, stored in a register, the initialization of the range could only be accomplished by entering the range through the DO statement. In the implementation proposed herein, the index variable and parameters *are* related to the similarly named variables in the rest of the program; the programmer can cause control to enter the range abnormally provided that the work of the initialization phase is simulated.

3. By the same reasoning, which allows the overriding of the second restriction, the value of the index will be available outside the range either after a normal completion or through a branch out without normal termination.

4. If the implementation of DO loops is not dependent on index registers, then the restriction of a unidirectional indexing is not necessary except that there may be a need to redefine the meaning of a DO statement. USASI Basic FORTRAN (Sec. 7.1.2.8) states that:

> The action succeeding execution of the DO statement is described by the following five steps:
>
> 1. The control parameter is assigned the value represented by the initial parameter. *This value must be less than or equal to the value represented by the terminal parameter.*[†]
> 2. . . .
> 3. . . . after execution of the terminal statement, the control variable . . . is *incremented* by the value represented by the associated incrementation parameter.
> 4. If the value of the control parameter after incrementation is *less than or equal to* the value represented by the associated terminal parameter, the action described in step 2 (executing the range) is repeated . . . If the value of the control variable is *greater than* the value represented by its

[†] Author's emphasis.

associated terminal parameter, the DO is said to be satisfied and *the control variable becomes undefined.*

If these restrictive specifications were revamped slightly, a standard DO statement could execute as specified but, at the same time, allow some desirable variations.

For example, if the restriction that the initial parameter value be less than or equal to the value of the terminal parameter is removed, then the relative values of the initial and terminal parameters are irrelevant and merely define a domain of values within which the DO range is to be executed. In paragraph 3 of Sec. 7.1.2.8, a negative incrementation should be allowed. In paragraph 4, the pertinent wording might be changed to read:

4. *If the value of the control parameter after incrementation is within the domain of the initial and terminal parameter values, the action described in step 2 . . . If the value of the control parameter is outside the domain of the initial and terminal parameters, the* DO *is said to be satisfied.*

The last statement, and *the control variable becomes undefined,* is purposely omitted, so that if the loop is terminated normally, the control variable will have the value at which the range was not repeated.

If the direction of the incrementation of the DO control variable (or index) is unknown at the time of compilation, the most efficient manner of testing of the control variable after incrementation is to call upon a system routine which will test the new value against the domain. In this case, the coding generated may take the form:

```
           LDA  J            Initial value to Index.
           ST   I
     NEXT  . . .
           . . .
           . . .
           LDA  I            Increment index.
           ADD  L
           ST   I
           ENA  *+2          Address of list to ACC.
           B    UNDO
                Address(I)
                Address(K)
                Address(L)
                NEXT
```

where the UNDO routine collects the addresses of the elements of the DO statement and the branching address from the appended list that follows the statement which transferred control to the routine. This routine checks the current value of the index against the limiting value, the test being influenced by the sign of the increment. That is, if the value of the increment is positive, the test in the UNDO routine is *less than or equal to* the limit, whereas if the sign is negative, the test is *greater than or equal to.*

5. At object time the DO statement is constructed so that the control variable can only be tested after a pass through the range; thus the range must be traversed at least once. However, under the redefinition proposed above, the initial value of the control parameter is a limit in the domain of values and hence is a valid value for a pass through the range. One may argue that if the direction of incrementation is determined at the entry to the range and is, by definition, unalterable, then the range should not be traversed if the successive incrementation would never reach the limiting value. For example,

```
DO 1 I=1,10,-1
```

should never cause a single execution of the range since the limit can never be attained. In view of the proposed rearrangement, above, the range would be executed with I=1 but not a second time with I=0 since this value of the control variable is outside the domain of the DO statement.

6. If the DO statement is implemented without the use of index registers, the elements do not need to be restricted to integer mode.[†] However, the compiler must be able to examine the mode and to link to an incrementation and testing routine of the appropriate mode.

7. Even when index registers are used, there is no reason to prohibit expressions as the elements of a DO list. For example, if the arithmetic statement scan routine is again considered as two routines, the left-hand side scanner and the right-hand side scanner, then each element in the DO list may be considered to be a right-hand side (executable), while the index (or control variable) is a left-hand side variable (or loadable). In each

[†] One reviewer has pointed out, however, that a reason for maintaining integer mode in a binary machine would be the difference, in operation, between the two statements:

```
DO 1 I = 0.01, 1.00, 0.01
```

and

```
DO 1 I = 1, 100, 1
```

case, when a DO statement is encountered, the expressions are evaluated and the results, if necessary, placed in temporary locations. For example,

```
DO 13 N(I+7)=K*J,N(3),I+3
```

would be compiled in the form:

```
       . . .
       . . .  }              Compute address of N(I+7).
       . . .
       ST    TEMP1           Store address in temporary location.
       LDA   K               K to ACC.
       MUL   J               ACC*J to ACC
       ST    -TEMP1          ACC to N(I+7)
       LDA   I               I to ACC
       INA   =3              ACC+3 to ACC
       ST    TEMP2 [†]
NEXT   . . .
       . . .
       . . .
ST13   LDA   -TEMP1          N(I+7) to ACC
       ADD   TEMP2           ACC+(I+3) to ACC
       ST    -TEMP1          ACC to N(I+7)
       SUB   Address(N(3))   Test against limit.
       BN    NEXT
       BZ    NEXT [‡]
```

In this type of implementation all the necessary auxiliary computation is performed prior to entering the DO range and therefore detracts from the main line program, not from the DO range itself. That is, the auxiliary computations are not executed each time through the range, thereby increasing the efficiency of the range. However, since these are one-time calculations, the values of the elements cannot be altered during the course of the execution of the range.

8. If the elements of the DO list are restricted to simple variables or constants and index registers are not used, there is no need to restrict the

[†] Note that at this point in the compilation the symbol table would contain a sublist appended to the entry referring to statement number 13, the elements in the sublist being the addresses

NEXT, -TEMP1, Address(N(3)), TEMP2

[‡] Note that for the purposes of this example, the UNDO routine was not used.

alteration of the elemental values during the execution of the range. However, if subscripted variables or expressions are permitted, and the addresses of subscripted variables and the values of expressions are computed outside the range in the initialization instructions, then these values are fixed and can only be altered by repeating the initialization computations. For example, if the implementation were to compute the expression or subscript *each* time the element was mentioned within the range, then such an allowance could be made, but the execution time would increase considerably. Further, the programmer would have to realize that such variations would not be in force until the next time through the range and not immediately after the value had been altered.

Suppose that the above example (DO 13 N(I+7)=K*J,N(3),I+3) were entered with I set to 3 and that during a traverse through the range, I was incremented by 1. Should one consider taking the control variable value from N(10) for the remainder of the range or from N(11) the value of which may already exceed the limiting value? Or since the value of the increment has been altered, should the value of the index be incremented by 1 immediately by implication of the change in incremental value or should such an alteration wait for the normal incrementation at the end of the range?

Hence there is some ambiguity as to the action to be taken when an elemental value is changed during the execution of the DO range. However, if the USASI rule book explained that alteration effects of elemental values are only noted during the next traverse of the range except when the value of the control variable is altered and its subscript is not, it would ease the problem of implementation and force the programmer to conform to that implementation.

Summary

The implementation of the ASSIGN and assigned GO TO statements in FORTRAN-like languages leads to many extensions of the language and possibly to some ambiguities in the syntactic analysis of the resultant statements. However, some strengthening of the rules regarding the inclusion of blanks in the statements would avoid these ambiguities. If, for example, it were stated that a blank is a delimiter, then many problems could be eliminated. For example, if the ASSIGN statement is implemented in its primitive form, it should be possible to include an ASSIGN statement that has a statement identifier variable in place of the statement number, such as ASSIGN I TO J. The exclusion of blanks from this statement would lead to ASSIGNITOJ which is not ambiguous, but what of

ASSIGN ITO TO TOT ?

The exclusion of blanks in this statement could lead the analyzer to extract the following forms:

ASSIGN I TO TOTOT
ASSIGN ITOTO TO T

or

ASSIGN ITO TO TOT

plus several statements which have too many second variables and thus be in error.

Similarly, it seems possible to implement, in the DO statement, a variable range execution by replacing the statement number in the defining statement by a statement identifier variable. The implementation of this feature, however, is best left to the experts.

6

INPUT/OUTPUT and FORMAT

The relationship between input/output statement lists and their corresponding FORMAT statement cannot be expressed in simple fixed terms. In particular, the control of data conversion either from internal mode to the output mode or from external form to internal mode is sometimes under the direction of the input/output statement, while on other occasions it is the responsibility of the chosen FORMAT statement. As a result, the manner in which input/output statements and FORMAT statements are compiled is closely related.

However, FORMAT statements cannot be compiled as a simple set of links to system routines since such a statement refers both to input and output. For example, a numeric specification (E,F or I) on input describes a transformation from external to internal mode, while on output the reverse is intended. On the premise that such translation routines are not themselves reversible, the FORMAT specifications will have to be interpreted at object time to ascertain the correct routine with which to link. Because of this need to interpret the FORMAT statement at object time, there is a question as to how far it is worth translating the FORMAT statement at compile time.

At best, the compile time translation of a FORMAT statement can only present the specifications in a form convenient for fast interpretation at object time. Further, if the compiler is to include in the object time routines, a routine to translate variable FORMAT, there must be a routine to translate source form FORMAT statements to this interpretive form. Hence, one may argue that all FORMAT statements should be translated and interpreted at object time. However, this has the disadvantage that object time is wasted in favor of saving compile time. Further, special steps need to be taken to ensure that source form FORMAT statements are not repeatedly translated when those statements ocur in the original source program. On the other hand, variable FORMAT statements must be repeatedly translated since the FORMAT source form character string may have been altered by programming and not only by a direct input statement. This argument assumes that the compiler has recognized that an array name has occurred in an input/output statement and has attached a tag to that array, this tag being set to one value whenever the array data are manipulated and to some other value when these data are translated by the FORMAT routine.

Even with such refinements, the accumulated object time in a series of production runs may be such that an alternative manner of translation and interpretation is necessary. Even though there may be a routine to translate source form FORMAT statements in the object time relocatable routines, the translation of source code statements still may be performed at compile time while variable FORMAT statements are translated at object time. This keeps object time translation time to a minimum, and only those programmers who utilize variable FORMAT will suffer from an increased execution time and a loss of available storage due to the inclusion of the variable FORMAT translation routine. The effect of such a compromise depends on the manner in which the total system is to be used. For example, in a computing center where few programs are generated and there is a substantial number of production runs, the saving of object time at the cost of compile time is important. On the other hand, in an educational or research environment, where the majority of runs are compile-run sequences, it is relatively unimportant as to whether FORMAT translation is performed at compile time or run time provided that the translation itself is programmed as efficiently as possible.

The Interacting Lists

Basically, input/output in FORTRAN consists of the manipulation and/or generation of a data string based on two lists, generated from the I/O statement and the set of FORMAT specifications. The I/O-generated list consists of a set of addresses, each referring to a variable, while the FORMAT list is a set of specifications. The data string may be either an input string waiting to be stored in internal form in memory or an empty string waiting to be filled from the memory. Associated with each list is a pointer which, for the purposes of this discussion, will be named:

SP	data string pointer
LP	I/O list pointer
FP	FORMAT specification pointer

These pointers will traverse each list under the control of the system input/output routine.

The FORMAT list is a set of link addresses with appropriate data so that the pertinent routines may be entered. Owing to the schizophrenic purpose of FORMAT statements, the link addresses merely point to another set of tests that divert the flow of control to a pair of routines which are relevant to input and output, respectively. However, since all data are converted in either direction, irrespective of the input or output physical unit, and all manipulation takes place in an internal data buffer area, these FORMAT routines are independent of the device used. The routines given in Table 6.1 are necessary.

Each of the TYPE routines in the table (ETYPE, FTYPE, etc.) has a direct influence on the string pointer (SP), whereas the other routines either direct the operation of the I/O unit (to empty or fill the buffer) or control the FORMAT pointer (FP) to pick up a new piece of data by a new specification.

In particular, LTPAR has no direct influence on the I/O action but rather prepares the RTPAR routine for possible reflection when the I/O list is longer than the FORMAT list. Hence the only action of the LTPAR routine is to place the current value of the FORMAT pointer (in fact, the address of the word to which the pointer is set) in the RTPAR routine. Similarly, the NLPAR routine must place the value of the number of repetitions and the return address in the NRPAR routine so that the repeated group may be traversed the correct number of times. Since repeated groups may be nested in a FORMAT statement, the number of repetitions and the addresses must

TABLE 6.1

Name	*Data Form* †	*Purpose*
ETYPE	WWDD	E-type specifications where WW = w and DD = d of E$w.d$
FTYPE	WWDD	F-type specifications where WW = w and DD = d of F$w.d$
ITYPE	WW00	I-type specification where WW is the field width of the specification.
ATYPE	WW00	A-type specifications, where WW is the w portion of the specification.
HTYPE	WWXX . . .	H-type specifications, where WW is the number of characters in the string and XX . . . is the character string. Note that in fact the actual string may be stored in BCD mode, and the number denoted by WW may be adjusted to reflect this internal size.
XTYPE	WW00	X-type specifications, where WW is the field width in nX.
REPET	WW00	Repeated specifications where WW = n of the forms nS$w.d$ or nTw, where S is either an E or F specification and T is an I or A.
NRPAR	—	To denote the right parenthesis of a repeated group of specifications.
NLPAR	WW00	A left parenthesis of a repeated group, where WW = n of $n(\ldots)$
SLASH	—	The / specification, i.e., end of record.
LTPAR	—	The left parenthesis of an unrepeated group.
RTPAR	—	The right parenthesis of an unrepeated group.

† This data form assumes a character orientation of the data words and restricts the field widths to two digits. Neither of these restrictions is necessary in practice.

be placed in a push down list with a last in, first out (LIFO) property. Provision must be made in this routine for ensuring that the list is renewed as each new FORMAT statement is used and that any list left over from an unsatisfied FORMAT is not carried over to the next statement.

The input/output list is also interpretive but contains facilities for in line object time computations. In particular, a system of tags and addresses is used to form the basis of an interpretive process, so as to facilitate the input and output of whole arrays and groups controlled by implied DO loops, the computation of subscripted variable addresses, and the checking of mode as well as to allow the possibility of outputting the result of an expression contained in the output list.

For example, the following items may be compiled:

Item in I/O list	*Tag*	*Address*
Simple integer variables and subscripted integer variables with constant subscripts	0	Address of variable
Simple real variables and subscripted real variables with constant subscripts	1	Address of variable

On the assumption that the interpreting routine can transfer control to inline instructions and that control can be returned to the interpreter by branching to a point in the interpreter named RENTR, variables subscripted by variables or expressions containing variables may be compiled by the inclusion of inline instructions. For example, if a tag of 2 is taken by the interpreter to mean that control is to be transferred to the instruction in the next word and the instruction

```
B     RENTR
```

is to transfer control back to the interpreter, the presence of subscripted variables may be compiled as:

```
2                 (tag)
. . .
. . .             Compiled instructions to compute
. . .             address of subscripted variable.
. . .
. . .
ST    *+4         Store address in interpretive list.
ENA   *+3         Reset list pointer.
ST    LP
B     RENTR
1     Address     (tag) Location of absolute address
                  of element to be input or output
                  after the execution of the inline
                  instructions.
```

Entire array references may be entered into the object interpretive list by the use of a special repeating tag. For example, a list containing an A that has been dimensioned as A(3,4) would compile to

```
1     Address(A(1,1))
3     0011
```

where the tag is assumed to be in the operation portion of the word and the address (or constant) in the operand position, and where a tag of 3 indicates that the address in the previous word is to be used as a base and incremented by unity 11 times so as to read in the remainder of the elements in the array. The number following the tag of 3 is one less than the number of elements in the array since the first element has already been referenced. The mode denoted by the first reference must, of course, be maintained throughout this operation. In this type of I/O operation, the order in which the elements are considered depends on the ordering of the elements in the data table. In the ordering proposed in Chapter 4, the elements are referenced in such a manner that the first subscript changes most rapidly and the last (or farthest right) most slowly.

Implied DO in I/O Lists

The same technique of leaving the interpretive mode may be utilized for the coding of implied DO loops within the I/O lists. Such a group may be recognized in a left to right scan by encountering an opening parenthesis. However, the compiler must scan the right-hand side of the group to obtain the variable name to be used as the implied DO index (control) variable. As a simple case, consider the I/O list

(I, I=1,N,2)

which compiles to the instructions and list:

```
     2
     ENA  1         I=1
     ST   I
NEXT ENA  *+3       Reset LP.
     ST   LP
     B    RENTR     Return to interpretive mode.
     0    I         List entry, integer variable I.
     2              Execute inline instructions.
     LDA  I         Increment I.
     INA  2
     ST   I
     SUB  N         Test for completion of loop.
     BZ   NEXT
     BN   NEXT
     ENA  *+3       Reset LP.
     ST   LP
     B    RENTR     Return to interpretive mode.
```

When the element in the group controlled by an implied DO is a subscripted variable dependent on the control variable of the DO, the oscillation between the inline coding and the interpretive mode can be extremely wasteful of execution time unless one recognizes at compile time that such an oscillation will occur and takes appropriate remedial action. For example, one can argue that, in general, elements are placed in an implied DO group since they are dependent on the control variable of the DO. Thus it may be anticipated that after one leaves the interpretive mode to initialize the DO control variable, inline instructions will need to be executed to compute the address(es) of subscripted variable(s). Similarly, if the I/O list contains more than one element in the implied DO group, it is wasteful to return to the interpretive mode to manipulate one variable address without checking whether the next set of instructions to be executed is also inline. To return from the inline mode to interpretive mode requires the execution of three instructions; the reverse motion requires one word of interpretive storage and the execution of innumerable instructions in the interpreter routine. Thus if the number of transfers of control to and from the interpreter can be minimized, the speed of execution may be enhanced. For example, consider the compilation of the I/O list:

```
(A(I),B(I+3,1),I=1,K,L)
```

as shown in Table 6.2.

The difficulty of compiling with economization results from the fact that the set of inline instructions must place the computed addresses into the interpretive list and the addresses into which these results are to be placed are not known when the inline coding is being generated; consequently, the object time loader must backfill these addresses. Further, when a group contains both variables dependent on the control index of the implied DO and "free" variables, all inline instructions should be compiled together and similarly all interpretive mode data kept in a single list.

With the ability to include inline instructions in an output list, it is not inconceivable to allow expressions in the list so that output data may be computed without storing the result in a storage location. In general, if every item in an output list is considered to be an expression (that is, an *executable*), then without appropriate tests in the I/O generator, the right-hand side scanner of the arithmetic statements may be utilized to generate both required inline instructions and addresses. However, when any single arithmetic operation is to be executed, the arithmetic generator will normally leave the result of the computation in the accumulator. Thus if control is

TABLE 6.2

Compile without economization				Compile with economization			
	2				2		
	ENA	1			ENA	1	
	ST	I			ST	I	
NEXT	ENA	*+3		NEXT	. . .		Compute address
	ST	LP			. . .		of A(I)
	B	RENTR			. . .		
	2				. . .		Compute address
	. . .		Compute address		. . .		of B(I+3,1)
	. . .		of A(I)		. . .		
	. . .				ENA	*+3	
	ENA	*+3			ST	LP	
	ST	LP			B	RENTR	
	B	RENTR			1	A(I)	
	1	A(I)			1	B(I+3,1)	
	2				2		
	. . .		Compute address		LDA	I	Increment I
	. . .		of B(I+3,1)		ADD	L	
	. . .				. . .		
	ENA	*+3			. . .		
	ST	LP					
	B	RENTR					
	1	B(I+3,1)					
	2						
	LDA	I	Increment I				
	ADD	L					
	. . .						
	. . .						

transferred back to the interpreter, this result may be lost. A special set of tags, such as 8 and 9, would indicate that the next word contained not the address of a variable but the actual value to be output. Only two tags are needed in this instance since no other inline operations can be performed. Thus the inline instructions to compute the value of an expression would place this result in the interpretive list behind the appropriate tag. Such a facility during input is meaningless, since an input list is a list of addresses in which data are to be stored and may be considered as a set of left-hand sides or loadables.

In the interpretive mode, one more code or tag is essential. Tag 2 transfers control from the interpretive mode to inline coding with the expectation that control will be returned; but a code is needed to signify the end of the I/O list and thereby to force an I/O action such as outputting the record created by a WRITE statement and reinitializing the interpreter and other associated routines in preparation for the next I/O statement execution.

IOPAK

The I/O list interpreter is merely a portion of the overall supervisor which must control all input/output by manipulating the pointers LP, FP and SP, combining the actions required by the I/O list and the FORMAT lists. The general aspects of this supervisor, which we shall name IOPAK, are shown in Fig. 6.1.

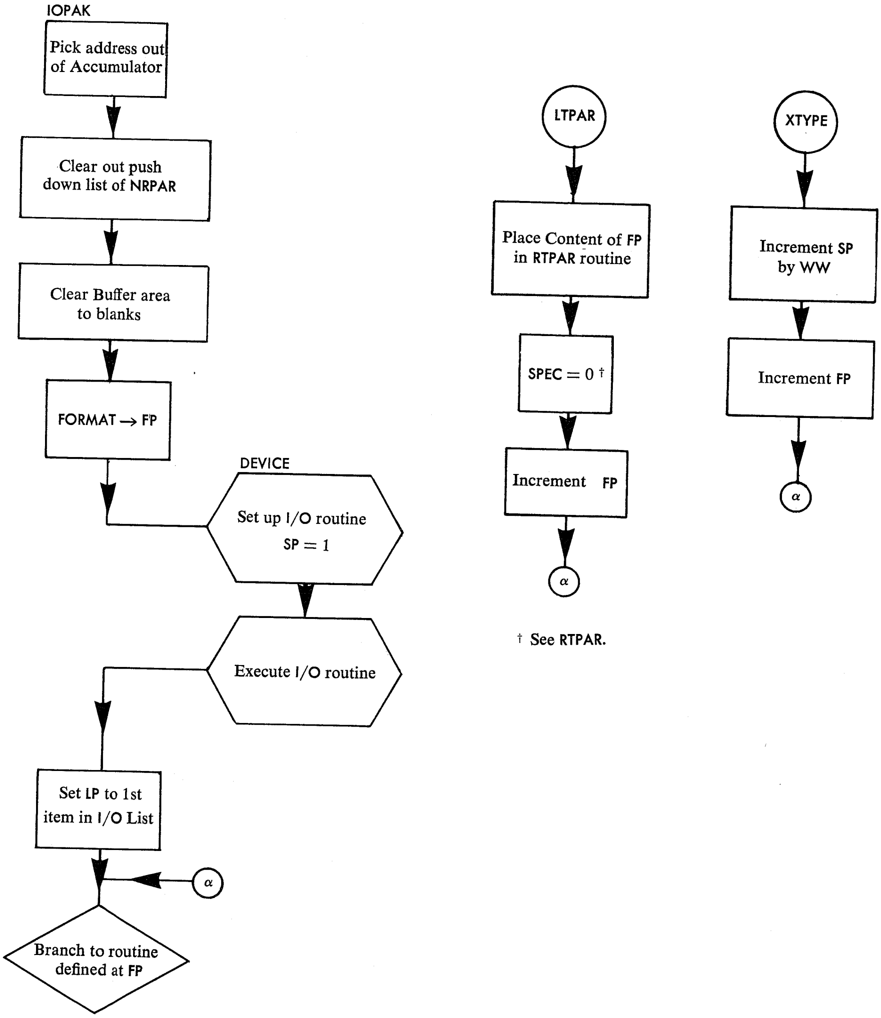
The diagram in Fig. 6.1 is based on the assumption that an I/O statement is compiled as:

```
ENA  *+2
B    IOPAK
     DEVICE      Code for device, i.e., unit number
     FORMAT      Address of first specification in FORMAT
. . .
. . .                        I/O interpretive list
. . .
```

Since some devices for input/output are bidirectional, a special tag is included in the device code to provide IOPAK with the information on the direction of the transference of data. The instruction to "Execute the I/O routine" appears at several points in the flow chart. Since the device code contains a tag specifying the direction of the action to be taken, some of these execution operations are bypassed. For example, if the operation is one of output, the execution of the I/O routine within the initialization phase is bypassed in favor of an output operation either when RTPAR or SLASH is reached, or when the output list of variables is exhausted in one of the TYPE routines. Similarly, the I/O routines in the TYPE routines are not executed on input, the input of the data having taken place during the initialization phase.

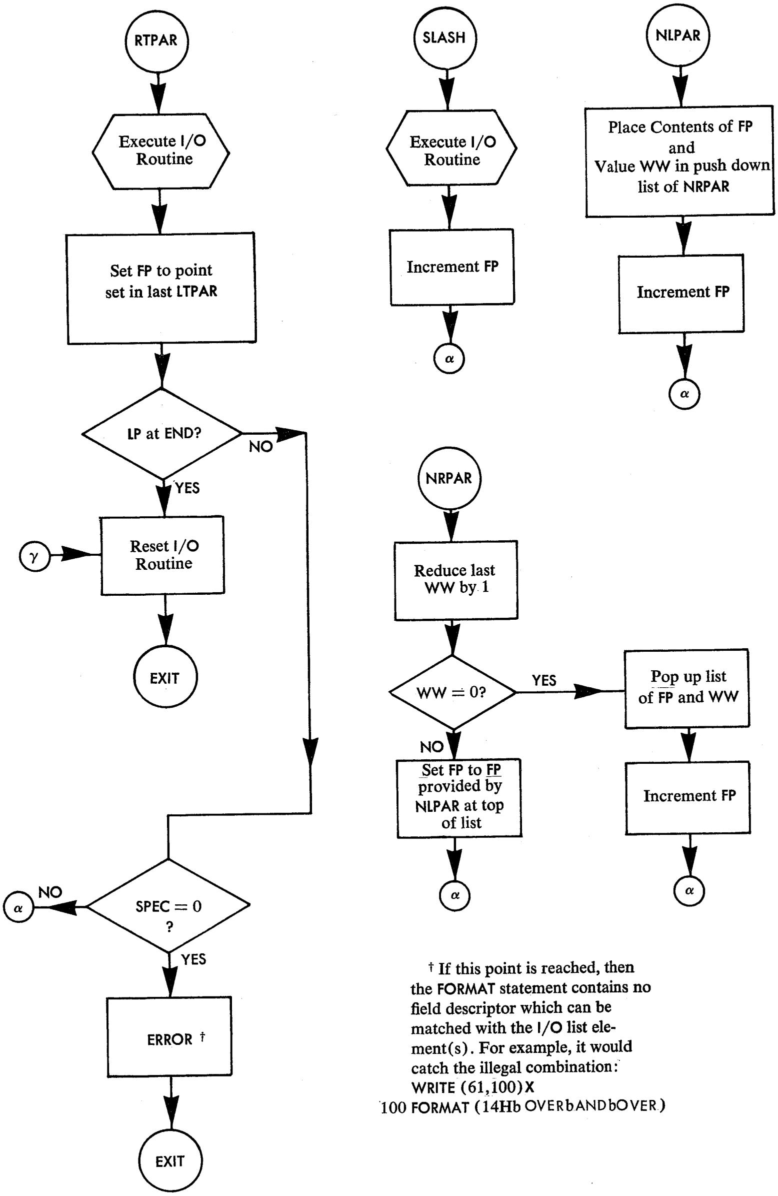
FORMAT in the above coding is the address of the referenced FORMAT statement, which will be used as the initial value of the FORMAT pointer, FP. The content of the accumulator on entry to IOPAK is the address of the first item in the I/O list which is, in fact, DEVICE.

In Fig. 6.1, several other routines are referenced. The I/O list interpreter has already been discussed. The I/O routine is possibly a single instruction into which the pertinent bits can be placed so as to execute the input to an internal buffer or to transfer the contents of the buffer to an output device. For example, in the IBM 1620, there are basically only two instructions pertinent to input and output: 37 XXXXX OZZOO for input, and 39 XXXXX OOZZO for output, where XXXXX is the address of the internal buffer area and ZZ is a code to reference a particular device. For example, ZZ=01 refers


IOPAK
Pick address out of Accumulator
Clear out push down list of NRPAR
Clear Buffer area to blanks
FORMAT → FP
DEVICE
Set up I/O routine SP = 1
Execute I/O routine
Set LP to 1st item in I/O List
α
Branch to routine defined at FP
LTPAR
Place Content of FP in RTPAR routine
SPEC = 0 †
Increment FP
α
XTYPE
Increment SP by WW
Increment FP
α

† See RTPAR.

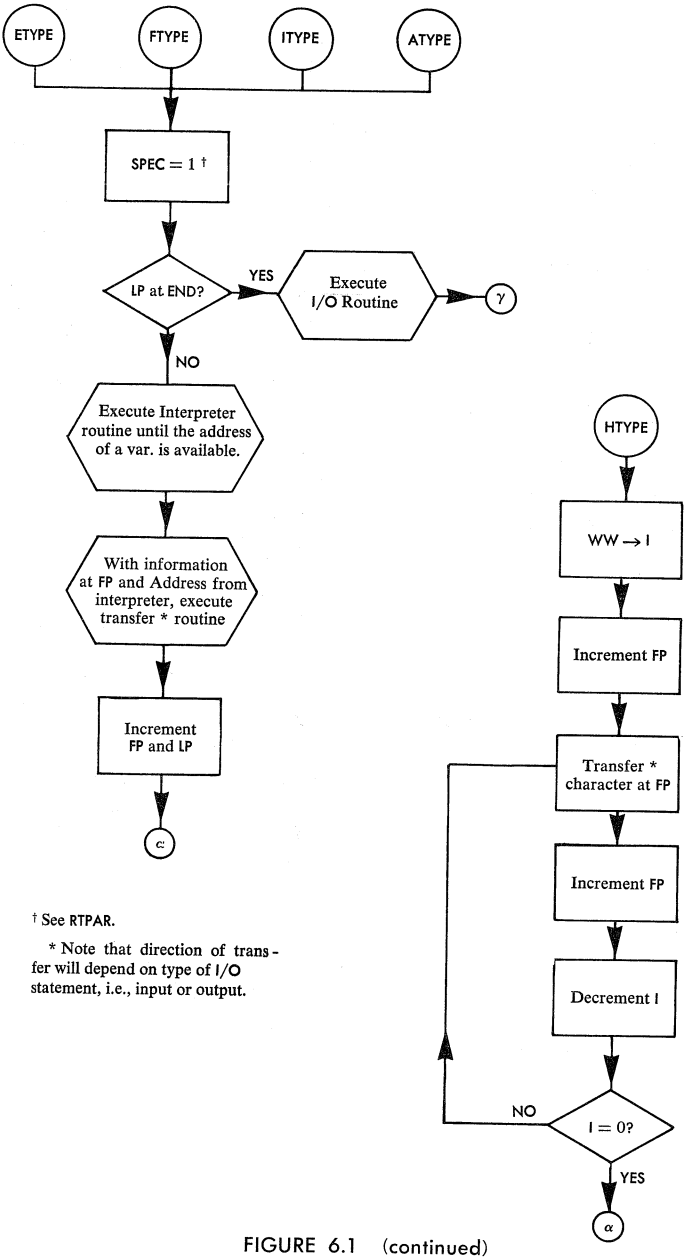
FIGURE 6.1



† If this point is reached, then the FORMAT statement contains no field descriptor which can be matched with the I/O list element(s). For example, it would catch the illegal combination:

```
    WRITE (61,100)X
100 FORMAT (14Hb OVERbANDbOVER)
```

FIGURE 6.1 (continued)


ETYPE
FTYPE
ITYPE
ATYPE
SPEC = 1 †
LP at END?
YES
Execute I/O Routine
γ
NO
Execute Interpreter routine until the address of a var. is available.
With information at FP and Address from interpreter, execute transfer * routine
Increment FP and LP
α
HTYPE
WW → I
Increment FP
Transfer * character at FP
Increment FP
Decrement I
NO
I = 0?
YES
α

† See RTPAR.

* Note that direction of transfer will depend on type of I/O statement, i.e., input or output.

FIGURE 6.1 (continued)

to the typewriter in both input and output, ZZ=02 refers to a paper tape punch, ZZ=03 to a paper tape reader, ZZ=04 to the card punch, and ZZ=05 to the card reader. Thus if a dummy instruction of the form 3Y XXXXX OZZOO is placed in IOPAK, the initialization portion of IOPAK may set up the instruction appropriate to the direction of transfer and the particular device. When a single internal buffer is used in the system, the address XXXXX is also fixed. However, if alternating buffers are used so that one may be filled while the other is being emptied, this address must also be filled at object time by IOPAK.

Within the E, F, I and A TYPE routines there is a reference to a transfer routine which is merely a routine that transfers data to or from the internal buffer, from or to the internal data areas, with conversion from or to BCD mode, if necessary.

When the I/O statement references a variable FORMAT statement that was read in at object time, reference must be made to the FORMAT translation routine before the IOPAK routine is entered, this FORMAT routine placing the translation into the area where the source form was stored. Thus to prevent the retranslation of this data by a second reference to the same variable FORMAT, a special tag should prefix the translation. Since some compilers allow the manipulation of the data in the variable FORMAT arrays and there is no restriction regarding the timing of this manipulation (that is, manipulation may take place after the use of that FORMAT specification), then the source form of the specification must be inviolate, the translation must be stored in a separate area, and the statement retranslated at each reference. A variable FORMAT input/output statement may be compiled as follows:

```
WRITE(5,A) . . .

             ENA  Address(A(1,1))        Address of array to ACC
             B    VARFM                  Branch to variable FORMAT
                                         translation routine

             ENA  *+2
             B    IOPAK
                  DEVICE
                  FORMAT
                  . . .
                  I/O list
                  . . .
```

Summary

The complications that arise at object time in executing the interacting I/O statements and their referenced FORMAT statements are similar to those connected with compiling the interacting statements COMMON, DIMENSION and EQUIVALENCE (see Chapter 4). However, the compilation of definition statements at compile time has a unique solution, whereas the object time execution of an input/output statement interacting with a FORMAT statement may lead to invalid combinations that must be routed out at object time. For example, it is difficult, without actually executing the statement of output, to ensure that a real variable is output under control of the appropriate specification. Since the specifications do not define the action to be taken in this type of situation, most compiler writers (and hence object code writers) make their own decision as to the appropriate action. In an educational system, this situation may be considered a fatal error, whereas in a commercial environment it may be intentionally decided that the appropriate mode conversion is to be executed when this anomaly is encountered. Thus a piece of data stated on the input medium as an integer may be stored as a real variable value without recourse to storing it as an integer and then including a special statement to change mode.

Similarly, the specification of a too narrow field width on output has no definite solution. One system will refuse to ouput the data, giving an object time error message, while another will output, without comment, only those low order characters that fit the specification.

Basic FORTRAN allows the writing of I/O statements without reference to a FORMAT statement, such a facility being a boon to lazy programmers and to computer science instructors who wish to overcome the complications of the FORMAT specification and acquaint their students with the core of the problem of algorithms as soon as possible. In this situation, the compiler must invent a FORMAT specification based on a standard set of specifications (i.e., E20.8, I20) from the I/O list storing this in the same manner as a written FORMAT statement or by adding special sections onto the I/O interpreter routine. In a completely free FORMAT system where the input data are not in any fixed FORMAT but merely arranged on the input medium with delimiters, such as blanks or commas, between each piece of data, the compiler cannot produce a FORMAT specification; however, it is feasible that the object time routines perform a double scan of each field, the first pass determining the field width which would then be provided to a FORMAT specification. The second scan would then return to the beginning of the field with this information and extract the data in the normal fashion.

The inclusion of expressions as valid elements in an output statement can lead to a considerable saving in execution time in a program outputting a great amount of information that is not required at later stages or that can be computed without logical decision instructions. In fact, with continuation cards it is quite possible for a program to exist that contains only a single instruction, except maybe STOP and END. For example, a complete table of trigonometric function values may be produced by the statement

```
PRINT,(FLOAT(I),SIN(FLOAT(I)),COS(FLOAT(I)),I=1,180)
```

by making use of a free FORMAT output statement and executables.

Problem

6.1 Redesign the flow chart in Fig. 6.1 to include free FORMAT input. As the key to this requirement in the compiled instructions, assume that free FORMAT is required when the address in the I/O list normally referring to the FORMAT statement is zero.

7

Polish String Notation

Arithmetic and logical expressions to be scanned for translation or analysis are, in general, in a form that is not suitable for scanning by a computer and translation to a set of machine language instructions. When one examines a statement by eye, it is comparatively easy to recognize groups or phrases and, for example, to locate the innermost parentheses. However, for the computer to locate this set of parentheses within a scanning process, the complete statement must be scanned more than once and possibly many times. The notation employed by a Polish logician, J. Lukasiewicz (and generally known as Polish String notation since most people can neither pronounce nor remember the name) is a means of representing arithmetic or logical expressions that is unambiguous, does not need parentheses to enforce the hierarchy of operations, and may be broken down to a set of operations and operands in a single scan. There are, in fact, several forms of Polish String notation. The particular form to be described in this chapter is known as "Reverse Polish." In this notation, each operator is preceded by its two operands. Thus the string $ab+$ is the Polish string form of the usual arithmetic expression $a+b$, it being assumed in this particular instance that each variable name is only one character in length.

Once an expression has been reduced to Reverse Polish, its translation back to the usual algebraic form is accomplished by scanning the string from left to right until the first operator is located. This is then placed between the two preceding operands, all three items becoming a parenthetical group or, in other words, a term in the string. For example, the string

$$xabc*d/+=$$

would be translated in the following steps:

1. In a left to right scan, the first operator is *, which is placed between the two preceding operands:

$$bc* \rightarrow (b*c)$$

The string becomes

$$xa(b*c)d/+=$$

and the term $(b*c)$ is regarded as an operand from this point on.

2. The scan is continued for the next operator to the right of the asterisk, which is /. This is placed between the term $(b*c)$ and the operand d:

$$(b*c)d/ \rightarrow ((b*c)/d)$$

3. The scan is continued further and the next operator is +. This is placed between the two preceding operands, and the string is transformed to

$$x(a+((b*c)/d))=$$

4. Finally, the last operator in the string is =; the completely transformed statement is

$$(x=(a+((b*c)/d)))$$

or

$$x = a + \frac{bc}{d}$$

The Conversion from Algebraic to Polish Notation

Performing the translation from normal algebraic notation to Reverse Polish is not quite as simple as the above procedure (which could be programmed easily), since normal notations are not explicit. For example, an order of execution dependent on the hierarchy of operators may be recognized in any algebraic statement, whereas Reverse Polish is independent of hierarchy. For example, the scalar expression

$$a + b(c-d)e - f$$

may be executed several ways, but with some implicit rules:

1. Parenthetical phrases are to be evaluated primarily.

2. Multiplication or division operators are to be executed before addition or subtraction.

3. When several operators of the same hierarchical level exist simultaneously in a statement, they are to be executed from left to right.

To overcome the handicaps of needing to refer to a set of rules of execution order, an expression that is to be translated should be fully parenthesized. That is, it should be written in a form that is a legal syntax of the rules:

<*operator*> := +|−|*|/|+
<*replacement sign*> := =
<*variable*> := *a*|*b*|*c*|*d*|*e*|*f*| . . . |*x*|*y*|*z*
<*constant*> :={<*digit*>}.{<*digit*>}|{<*digit*>}
<*digit*> := 1|2|3|4|5|6|7|8|9|0
<*expression*> := <*variable*>|<*constant*>|
(<*expression*><*operator*><*expression*>)|
(<*sign*><*expression*>)
<*sign*> := +|−
<*statement*> := <*variable*><*replacement sign*><*expression*>

If these rules are used, no pair of parentheses contains more than one operator at the same parenthetical level, where the parenthetical level of an operator is defined as the number of parentheses between it and the outside of the statement. For example, the parenthetical level of each operator is written beneath each operator in the following statement:

$$(x \underset{1}{=} (a \underset{2}{+} ((b \underset{4}{*} c) \underset{3}{/} d)))$$

Once an expression has been fully parenthesized, it may be translated to Reverse Polish by working from the innermost parenthetical phrase toward the outside of the statement. Taking the above example, the innermost parenthetical group is

$$(b * c)$$

which becomes

$$bc*$$

Let us use { . . . } to distinguish that portion of the expression which has been translated and may now be regarded as a single term; then the above statement becomes:

$$(\, x \underset{1}{=} (\, a \underset{2}{+} (\, \{bc*\} \underset{3}{/} d\,)))$$

The symbol / appears at the next level:

$$(\, x \underset{1}{=} a \underset{2}{+} \{bc*d/\}\,))$$

Continuing to level 2:

$$(\, x \underset{1}{=} \{abc*d/+\}\,)$$

and, subsequently, at the lowest level:

$$\{xabc*d/+=\}$$

Since no other untranslated parenthetical groups remain, the { . . . } may be dropped. Translation from an arithmetic statement to Reverse Polish does not necessarily result in a unique string. The result depends on the parenthesizing. Automatic parenthesizing routines that recognize the hierarchy of a left-most operator in a string of equal hierarchical level operators will produce the same string consistently, whereas manual parenthesizing which takes advantage of the associativeness of + and * can produce differing Reverse Polish strings. For example, the algebraic string

$$a(b-c)d$$

may be parenthesized to two manners:

$$((\, a * (\, b - c\,))\, * d\,) \qquad \text{and} \qquad (\, a * ((\, b - c\,) * d\,))$$

which translate to

$$abc-*d* \qquad \text{and} \qquad abc-d**$$

respectively. A more startling result is found when a normal string contains many operators of the same level:

$$a + b + c + d$$

$$(((a + b) + c) + d) \qquad (a + (b + (c + d)))$$

$$ab{+}c{+}d{+} \qquad abcd{+}{+}{+}$$

Although a pair of parentheses may not contain more than one operator at the same level as the parentheses themselves (that is, no more than one unparenthesized operator), there may be as many as two operators at the next level, three at the next, and so on. For example, the statement

$$a + b * c + d + e * f$$

may be wholly parenthesized in the form

$$((a \underset{2}{+} (b \underset{3}{*} c)) \underset{1}{+} (d \underset{2}{+} (e \underset{3}{*} f)))$$

Thus in translation to Reverse Polish, there are two possible starting points for the conversion process: the two operators at level 3. In fact, the order of conversion is irrelevant, since all orders of hierarchy are dissipated by the inclusion of parentheses. However, no operator can be involved in a conversion process until all other operators of higher parenthesis level have been eliminated. In a perfect situation, it would be advisable to convert all equal level operators with their operands to Reverse Polish simultaneously. For example,

$$(a * b - (c + d)/ e) * f$$

$$(((a \underset{3}{*} b) \underset{2}{-} ((c \underset{4}{+} d) \underset{3}{/} e)) \underset{1}{*} f)$$

$$(((a \underset{3}{*} b) \underset{2}{-} (\{cd{+}\} \underset{3}{/} e)) \underset{2}{*} f)$$

$$((\{ab*\} \underset{2}{-} \{cd{+}e/\}) \underset{1}{*} f)$$

$$(\{ab*cd{+}e/{-}\} \underset{1}{*} f)$$

$$\{ab*cd{+}e/{-}f*\}$$

The unary operators, and, in particular, the unary minus, must hold a special place in Reverse Polish notation since only one operand is associated with this operator. Thus, whereas the production rule of converting from Reverse Polish binary operators to normal notation may be described as

$$<\mathit{operand}><\mathit{operand}><\mathit{binary\ operator}> \rightarrow$$
$$(<\mathit{operand}><\mathit{binary\ operator}><\mathit{operand}>)$$

the production rule to be used in connection with a unary operator is

$$<\mathit{operand}><\mathit{unary\ operator}> \rightarrow (<\mathit{unary\ operator}><\mathit{operand}>)$$

Another drawback to the use of unary operators (which, incidentally, are merely a shorthand method of denoting an operation involving the zero operand) is the use of the same symbolism for unary plus and minus as binary plus and minus, respectively. A unary operator may be recognized because:

(a) It is the first character in an algebraic string, for example,

$$-a - b$$

(b) It is the first character following an opening parenthesis, for example,

$$a * (-b)$$

In the following discussion, the unary plus will be ignored, and a unary minus will be represented by the special symbol ~. Thus in the translation, $(-a)$ becomes $\{a^\sim\}$. According to the rules of total parenthesizing, a unary minus must be surrounded by parentheses; thus $((-a) * (-b))$ becomes $(\{a^\sim\} * \{b^\sim\})$ and, finally, $a^\sim b^\sim *$.

Problems

7.1 Convert the following Reverse Polish strings to normal algebraic strings:

(a) $a{\sim}cd{*}{+}bc{-}/zk{\sim}{\uparrow}{+}$
(b) $xcd{\uparrow}e/ab{*}{+}{=}$
(c) $ab{+}cd{*}/egf{\sim}{+}{*}{-}{\sim}$

7.2 Convert the following algebraic statements to Reverse Polish notation:

(a) $-a + b * c * d / e^{f}$
(b) $(a - b - c)/ d / e$
(c) $a(b/c + d/e) * f(-g - h)$

Expressions Including Unary Operators

Certain combinations of operators and the unary minus may be manipulated so that the number of unary minus operators are minimized in an expression. In particular, transformations (or productions) using π and ρ terms (listed below) may be used to manipulate expressions.

	Original expression			*Expanded expression*
1.	$\pi + (-\rho)$			$\pi - \rho$
	$\pi\rho^{\sim} +$			$\pi\rho-$
2.	$\pi - (-\rho)$			$\pi + \rho$
	$\pi\rho^{\sim} -$			$\pi\rho+$
3.	$\pi * (-\rho)$			$-(\pi * \rho)$
	$\pi\rho^{\sim} *$			$\pi\rho*^{\sim}$
4.	$\pi / (-\rho)$			$-(\pi / \rho)$
	$\pi\rho^{\sim} /$			$\pi\rho/^{\sim}$
5.	$-\pi - \rho$			$-(\pi + \rho)$
	$\pi^{\sim}\rho -$			$\pi\rho+^{\sim}$
6.	$-\pi + \rho$	or	$-(\pi - \rho)$	$\rho - \pi$
	$\pi^{\sim}\rho+$	or	$\pi\rho-^{\sim}$	$\rho\pi-$
7.	$-\pi * \rho$			$-(\pi * \rho)$
	$\pi^{\sim}\rho *$			$\pi\rho*^{\sim}$
8.	$-\pi / \rho$			$-(\pi / \rho)$
	$\pi^{\sim}\rho /$			$\pi\rho /^{\sim}$

and of course:

9.	$(-(-\pi))$	π
	$\pi^{\sim\,\sim}$	π

Since both π and ρ represent terms in the productions listed, identities 5 to 8 can only be used when the unary minus is the operator farthest to the right in a string of operators. Hence when any expansion of a string is to be manipulated so as to remove unary minus operations, the first process must be to move the imbedded unary minus operators to the right-most position in an operator string. For example, consider

$$((-b)*(-a))$$

that is, $b^{\sim}a^{\sim}*$ in Reverse Polish notation. From production 3, the unary minus in the pair of operators farthest to the right may be interchanged with the multiplication operator; that is, the string becomes $b^{\sim}a*^{\sim}$. If braces are added for clarity, it may be seen that the first four characters in the string are similar to the starting formula in step 7:

$$\{b^{\sim}a*\}^{\sim}$$

and thus may be manipulated to the string

$$\{ba*^{\sim}\}^{\sim}$$

When the parentheses are removed, the expression is similar to the starting formula of production 9, where the two unary minus operations cancel. Thus the resultant string is $ba*$, or in normal form $b*a$.

Consider the expression

$$-d-a+b*(-c)$$

that is, in Reverse Polish,

$$d^{\sim}a-bc^{\sim}*+$$

In the following manipulation, the underscored operators have been either manipulated or generated in a production.

From production 3:

$$d^{\sim}a-bc\underline{*^{\sim}}+$$

From production 1:

$$d^{\sim}a-bc*\underline{-}$$

Let us now bracket the groups for clarity:

$$\{d^{\sim}a-\}\{bc*-\}$$

From production 5:

$$\{da\underline{+\}^{\sim}}\{bc*\}-$$

Again production 5:

$$\{da+bc*\}\underline{+^{\sim}}$$

At this point no more unary minus operators remain inside, and the string may be converted back to the usual form:

$$-(d+a+b*c)$$

In connection with the involution operator, two further productions are necessary:

10.	$a*b\uparrow(-c)$	$a/(b\uparrow c)$
	$abc^{\sim}\uparrow*$	$abc\uparrow/$
11.	$a/(b\uparrow(-c))$	$a*b\uparrow c$
	$abc^{\sim}\uparrow/$	$abc\uparrow*$

Problem

7.3 Reduce or eliminate the unary minus operators in the following scalar algebra strings:

(a) $-\left[\frac{a}{-b}+(-c)\right]$

(b) $(-ab+c)^{-(d-e)}$

(c) $-(a-(c*(-b/(d^{-k}))))$

Parenthesizing Expressions

The translation from normal algebraic statements to Reverse Polish notation depends on the existence of a fully parenthesized statement. Since this is not the form in which statements are generally presented, let us consider an algorithm for the parenthesizing of unparenthesized or partially parenthesized statements. Basically, the parenthesizing of an expression defines the order of execution of each operator without regard for the hierarchy of these operators. However, the technique of parenthesizing must depend on this hierarchy. If each expression is delimited by two pseudo-operators, ⊢ and ⊣, which define the beginning and end of the statement, respectively, then the hierarchy levels may be tabulated as follows:

⊢ ⊣	0
=	1
+ −	2
* /	3
↑	4
~	5

The parenthesizing process consists of the following steps:

1. Add the beginning and end delimiters to the statement.

2. Place the operator hierarchy values under each operator. Note, we shall use the term $V(\underset{i}{\text{op}})$ for the hierarchical value of operator i, where i is the position number of the operator from the left-hand end of the statement. That is, the beginning operator ⊢ always has a position number 1.

3. Starting from the left-hand end of the statement, $(\text{op}_1 = \vdash)$, set the counter at $n = 1$.

4. Scan along the statement to the right until the hierarchical value of the operator n is found to be greater than or equal to operator n+1, that is,

$$V(op_n) \geqq V(op_{n+1})$$

5. Place a closing parenthesis to the left of op_{n+1}.

6. From op_n move left until op_i is found, so that

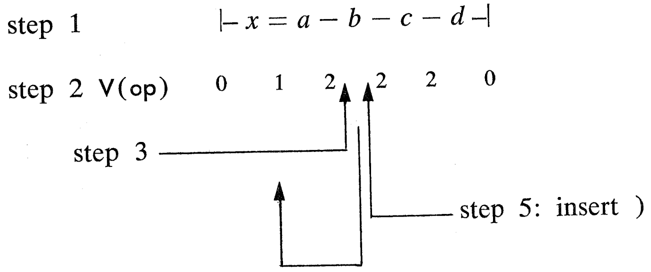
$$V(op_i) \leqq V(op_n)$$

7. Place an opening parenthesis to the right of op_i.

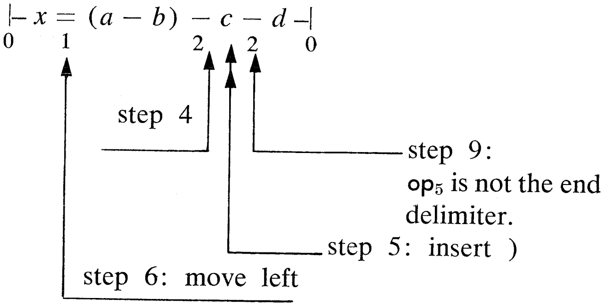
8. Remove $V(op_n)$.

9. If op_{n+1} is the end delimiter and op_{n-1} is the beginning operator, then the parenthesizing is complete. Otherwise, set the pointer to op_{n-1} (that is, decrease the value of the pointer by 1). If $V(op_n) \geqq V(op_{n+1})$, go to step 5; otherwise, go to step 4.

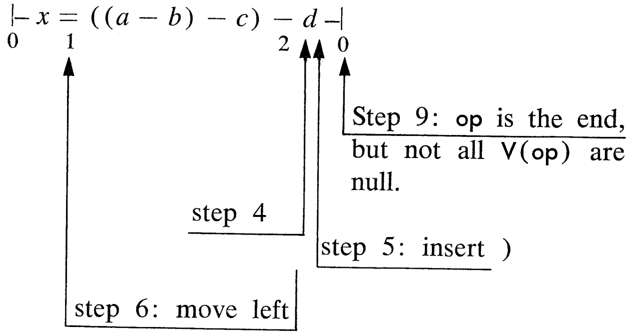
For example, consider the delimited string:



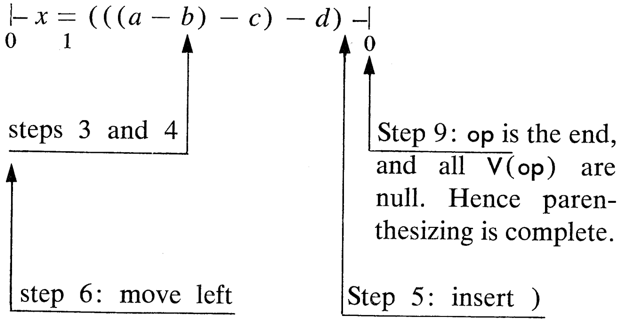
step 6: move left
step 7: insert (
step 8: remove $V(op_3)$
step 9: op_4 is not the end delimiter.


$\vdash x = (a - b) - c - d \dashv$
0 1 2 2 0
step 4
step 9:
op_5 is not the end delimiter.
step 5: insert)
step 6: move left

step 7: insert (
step 8: remove $\mathsf{V}(\mathsf{op}_4)$


$\vdash x = ((a - b) - c) - d \dashv$
0 1 2 0
Step 9: op is the end, but not all V(op) are null.
step 4
step 5: insert)
step 6: move left

step 7: insert (
step 8: remove $\mathsf{V}(\mathsf{op}_5)$


$\vdash x = (((a - b) - c) - d) \dashv$
0 1 0
steps 3 and 4
Step 9: op is the end, and all V(op) are null. Hence parenthesizing is complete.
step 6: move left
Step 5: insert)

step 7: insert (
step 8: remove $\mathsf{V}(\mathsf{op}_2)$

The resulting string is:

$$\vdash (x = (((a - b) - c) - d)) \dashv$$

In the following example, the individual steps are not shown, but an arrow indicates the operator chosen in step 4.

$$\vdash x = a * b / c * d \dashv$$

$$\underset{0}{\vdash} x \underset{1}{=} a \underset{\underset{\uparrow}{3}}{*} b \underset{3}{/} c \underset{3}{*} d \underset{0}{\dashv}$$

$$\underset{0}{\vdash} x \underset{1}{=} (a * b) \underset{\underset{\uparrow}{3}}{/} c \underset{3}{*} d \underset{0}{\dashv}$$

$$\underset{0}{\vdash} x \underset{1}{=} ((a * b) / c) \underset{\underset{\uparrow}{3}}{*} d \underset{0}{\dashv}$$

$$\underset{0}{\vdash} x \underset{\underset{\uparrow}{1}}{=} (((a * b) / c) * d) \underset{0}{\dashv}$$

$$\underset{0}{\vdash} (x = (((a * b) / c) * d) \underset{\underset{\uparrow \ \text{Complete}}{0}}{\dashv}$$

If a statement is already partially parenthesized, the above procedure must be modified slightly. In particular, if a left parenthesis is encountered during a left to right scan, the string following that operator up to the closing right parenthesis may be regarded as an entirely separate statement. Once that inner statement has been parenthesized, the outer statement may be parenthesized. For example, after the inner groups of the string:

$$\underset{0}{\vdash} x \underset{1}{=} (a \underset{2}{+} b \underset{2}{+} c) \underset{3}{/} (c \underset{2}{+} d \underset{3}{*} e) \underset{0}{\dashv}$$

have been parenthesized, the statement is of the form

$$\underset{0}{\vdash} x \underset{1}{=} (((a + b) + c)) \underset{3}{/} ((c + (d * e))) \underset{0}{\dashv}$$

After the next two steps, the parenthesizing of the multiplication and replacement operators, respectively, the statement takes the form

$$\vdash (x = ((((a + b) + c)) / ((c + (d * e))))) \dashv$$

This technique adds an overabundance of parentheses since each group within a parenthesis pair is reparenthesized. The technique may be improved by using the opening and closing parentheses to alter the hierarchical order or level of operators. For example, one may add 10 to all V(op)s when a left parenthesis is encountered, and subtract 10 when a right parenthesis is found. At the same time, parentheses are dropped (or omitted) from the string. When, in the above example, hierarchy values are added as affected by parentheses and the parentheses are dropped, the expression becomes

$$\begin{array}{ccccccccccccccc} \vdash & x & = & a & + & b & + & c & / & c & + & d & * & e & \dashv \\ 0 & & 1 & & 12 & & 12 & & 3 & & 12 & & 13 & & 0 \end{array}$$

This scheme eliminates the necessity to make an exception to the algorithm for the recognition of parentheses and also provides two other features.

1. Unpaired parentheses exist in the statement when the value of the level of the end delimiter is not zero.

2. Extra unneeded parentheses are eliminated. For example, if in the statement:

$$x = (a) + b \,/\, (c)$$

the level values are added and the parentheses dropped,

$$\begin{array}{ccccccccc} \vdash & x & = & a & + & b & / & c & \dashv \\ 0 & & 1 & & 2 & & 3 & & 0 \end{array}$$

parenthesizing gives:

$$\vdash (x = (a + (b \,/\, c))) \dashv$$

Problem

7.4 Given the following operator hiararchy table (from lowest level to highest),

$\vee$ (or)

$\wedge$ (and)

$< \; > \; \leq \; \geq \; = \; \neq$

~ (not)

$+ \; -$ (both binary)

$* \; /$

$\uparrow$

$+ \; -$ (both unary)

functions

parenthesize the following expressions:

(a) $y > 3 \vee b$

(b) $x \uparrow 2 - |x * y| > y \,/\, x$

(c) $a \wedge b \vee c \wedge d \neq {\sim}(a \vee b)$

(d) $\sin(x) \uparrow 2 + \cos(x) \uparrow 2 = 1$

(e) $\sinh(x) = (\exp(x) - \exp(-x)) \,/\, 2$

A parenthesizing process of particular interest is that utilized in the FORTRANSIT system of the IBM 650. In this process, the hierarchical table is used as a basis for the insertion of parentheses around each operator. If an operator and its operands are to be enclosed properly in parentheses, the highest level operator must be imbedded most deeply in a nested set of parentheses and the lowest operator enclosed by the fewest parentheses. Thus the FORTRANSIT parenthesizing routine places opposite-facing parentheses about each operator, the number of parentheses being in the inverse ratio to the hierarchial level. Consider the set of expressions that may be formulated from the restricted set of operators with the following hierarchical levels:

(binary)	+ −	level 1,	inverse level 3,
	* /	2,	2,
	↑	3,	1.

Consider the particular expression

$$a * b + c$$

If a set of parentheses is placed around each operator so that the number of parentheses is equal to the number of the inverse level, the expression becomes

$$a\,))*((\,b\,)))+(((\,c$$

Obviously, this expression is invalid since the number of parentheses that closes a group exceed the number that opens a group at at least one point in the string. To overcome this, the opening and closing delimiters must be added with (in this case) a hierarchical level of 0 and an inverse level value of 4. Further, since delimiters are one sided, parentheses need only be added between the delimiter and the expression. The above expression then becomes

$$((((a\,))*((\,b\,)))+(((\,c\,))))$$

At this point, an overabundance of parentheses has been created, making i difficult to determine whether or not the parenthesizing is complete or sufficient. If the matching parentheses surrounding each operand are removed the expression is reduced to

$$((a * b) + c)$$

When operators of the same hierarchical level occur in sequence, this simplified system breaks down. For example,

$$a - b - c$$

is parenthesized to

$$((((a))) - (((b))) - (((c))))$$

and reduces to

$$(a - b - c)$$

which is not satisfactory. Thus when the operator being surrounded is o the same level as that parenthesized immediately beforehand, the inverse levels in the whole table must be increased by 1, and an extra parenthesis added to the opening delimiter. Thus the above expression is parenthesized to the string

$$(((((a))) - (((b)))) - ((((c)))))$$

which reduces to

$$((a - b) - c)$$

Consider the expression

$$a + b + c \uparrow d - e$$

which parenthesizes to

$$(((((a))) + (((b)))) + ((((c)) \uparrow ((d)))) - ((((e)))))$$

Canceling the matching parentheses around each variable reduces the expression to

$$((a + b) + ((c \uparrow d)) - e)$$

This reduced expression (which still has two extra parentheses surrounding the involution operator) still has two operators at the same level, namely an add and a subtract. This is because the intervening involution operator masked the fact that, in the order of execution, the add and subtract would be in sequence. To overcome this object, the rule regarding the increase in the hierarchical level (inverse) should be altered to read that the inverse hierarchical level should be increased after each subsequent usage of the same operator level. That is, the inverse hierarchical level should be increased at the second and all subsequent occurrences of an operator from a single hierarchical level. With this rule the above expression would parenthesize to the following string:

$$((((((a))) + (((b)))) + ((((c)) \uparrow ((d))))) - (((((e))))))$$

which would reduce to

$$(((a + b) + ((c \uparrow d))) - e)$$

and further to

$$(((a + b) + (c \uparrow d)) - e)$$

This particular technique is an interesting approach to the problem of parenthesizing, but as will be shown later is a redundant operation in the arithmetic generator. However, in those specialized systems (such as FORMAC) where parenthesizing is one of the primitive operations, it may be worthwhile.

Problems

7.5 Extend the above algorithm to include the occurrence of parentheses in the original expression.

7.6 Write a program to read in a partially parenthesized expression, to parenthesize the expression, and to output the result. The program may include the replacement sign, unary operators and the operators +, —, *, / and ↑; each variable consists of a single letter, and no constants are included. If the FORTRANSIT method is used, discard any redundant parenthesis pairs.

Direct Conversion

After a statement is parenthesized, the next stage of translation is that described previously for converting a fully parenthesized expression to Reverse Polish notation. Let us now consider the techniques of combining these two processes. The process of converting from a parenthesized group to Reverse Polish notation may be described by the production

$$(<\textit{operand}><\textit{operator}><\textit{operand}>) \rightarrow \{<\textit{operand}><\textit{operand}><\textit{operator}>\}.$$

Thus if one knows where the parentheses are to be added in a string, that group may be immediately converted to Reverse Polish notation. However, the FORTRANSIT parenthesizing routine is not satisfactory for this purpose, since the parentheses are added before the groups are defined.

Using the parenthesizing routine discussed on page 170, we may proceed to step 4, locating the highest level operator before altering the algorithm It may be shown that in the first pass of the scan, the chosen operator i adjacent to its operands which are simple variables. Thus, these three item may be converted to Reverse Polish notation immediately. In fact, th operator and right-hand operand merely change places, and the three item form a new operand. If, for the purposes of demonstration, the operato and operands currently being converted to Reverse Polish notation are enclosed by { and }, then, unless one or other of the operands is already enclosed, the process of conversion may be described as follows:

1. Choose the operator and operands to be converted (by the same rule used for the parenthesizing algorithm) and surround each operand and th operator by { and }.

2. Within the string, exchange the positions of the operator and its right hand operand, carrying the enclosing braces with each item.

3. Scan the string and remove any adjacent braces. After this operation only two braces will remain in the string. These braces will surround th converted string which is considered henceforward as an operand.

4. Repeat the above steps until the string is totally enclosed in braces at which point the string has been converted.

Consider the string

$$\begin{array}{ccccccccc} \vdash & x & = & a & + & b & / & c & \dashv \\ 0 & & 1 & & 2 & & 3 & & 0 \\ & & & & & & \uparrow & & \end{array}$$

where the arrow denotes the first chosen operator by the rules of the parenthesizing algorithm. Enclosing the operator and its operands in braces and exchanging the positions of the operator and right-hand operand, we obtain

$$\begin{array}{cccccc} \vdash & x & = & a & + & \{b\}\ \{c\}\ \{/\} & \dashv \\ 0 & & 1 & & 2 & & 0 \end{array}$$

Cancel the adjacent braces

$$\begin{array}{ccccccc} \vdash & x & = & a & + & \{bc/\} & \dashv \\ 0 & & 1 & & 2 & & 0 \end{array}$$

The next operator to be included in the grouping is the addition sign whicl has a left-hand operand of *a* and a right hand operand *bc/*. When brace are added to the string, the right-hand operand is not affected since it is already enclosed. At this stage the string is of the form

$$\underset{0}{\vdash} x \underset{1}{=} \{a\}\ \{+\}\ \{bc/\} \underset{0}{\dashv}$$

Exchanging the positions of the right-hand operand and the operator gives:

$$\underset{0}{\vdash} x \underset{1}{=} \{a\}\ \{bc/\}\ \{+\} \underset{0}{\dashv}$$

Removing the adjacent braces:

$$\underset{0}{\vdash} x \underset{1}{=} \{abc/+\} \underset{0}{\dashv}$$

The last three steps are

$$\underset{0}{\vdash} \{x\}\ \{=\}\ \{abc/+\} \underset{0}{\dashv}$$

$$\underset{0}{\vdash} \{x\}\ \{abc/+\}\ \{=\} \underset{0}{\dashv}$$

$$\underset{0}{\vdash} \{xabc/+=\} \underset{0}{\dashv}$$

An input string containing embedded parentheses may be handled in the same manner provided the hierarchical levels are adjusted within the parenthetical groups and the parentheses dropped before the conversion is attempted. For example, consider the input string

$$x = (a + b)\ /\ (c \uparrow d)$$

Using a bias of 10 within the parenthetical groups, and dropping the parentheses, we obtain the input string:

$$\underset{0}{\vdash} x \underset{1}{=} a \underset{12}{+} b \underset{3}{/} c \underset{\substack{14 \\ \uparrow \\ \text{chosen} \\ \text{operator}}}{\uparrow} d \underset{0}{\dashv}$$

The steps in the conversion process are

$$\underset{0}{\vdash} x \underset{1}{=} a \underset{12}{+} b \underset{3}{/} \{cd\uparrow\} \underset{0}{\dashv}$$

$$\underset{0}{\vdash} x \underset{1}{=} \{ab+\} \underset{3}{/} \{cd\uparrow\} \underset{0}{\dashv}$$

$$\underset{0}{\vdash} x \underset{1}{=} \{ab+cd/\uparrow\} \underset{0}{\dashv}$$

$$\underset{0}{\vdash} \{xab+cd\uparrow/=\} \underset{0}{\dashv}$$

Summary

Polish string notation is important not only because it is a system that transcends the need for a hierarchy of operators and hence does not need parentheses, but also because the order of concatenation of variables and operators defines an order of execution of the expression. According to the rules of parenthesizing, the first operator and operands to be converted to Reverse Polish notation from an algebraic normal notation string are those of the first innermost parenthetical group, which is also the first group to be computed during execution.

Hence, in the same manner that the algorithm to convert from normal notation to a fully parenthesized form and the algorithm to convert from the parenthesized form to Polish notation were amalgamated, a single algorithm can be created to convert the normal algebraic mode to a set of computer instructions, once the relation between Reverse Polish notation and computer instructions is established. This is the topic of the next chapter.

8

The Ascan Generator

The previous chapter discussed the techniques and algorithms for the generation of Reverse Polish strings from more normal algebraic statements. In this chapter this concept will be extended to produce a set of machine instructions. This discussion will be based on those instructions already defined for an imaginary machine, together with new instructions which will be added to the repertoire as needed. We shall not discriminate between the various modes of execution (that is, real, integer, etc.) since these have no part in the general algorithm and are either machine or language dependent. For the moment, let us review the arithmetic instructions and their semantics:

Instruction	*Meaning*
LDA a	$a \rightarrow$ ACC
ADD a	ACC $+ a \rightarrow$ ACC
SUB a	ACC $- a \rightarrow$ ACC
MUL a	ACC $* a \rightarrow$ ACC
DIV a	ACC $/ a \rightarrow$ ACC
ST a	ACC $\rightarrow a$

The conversion from normal algebraic notation to Reverse Polish notation is unhampered by the need to convert the symbolism of the string. That is, the operands and operators have the same meaning in a normal algebraic string as in a Reverse Polish string. However, when one converts to machine instructions, the symbolism is different (for example, + becomes the mnemonic ADD), and more than a single string is created. In general, the conversion of a single set of operands and operator in the Reverse Polish notation to a set of machine instructions may be represented by the production

$$\begin{array}{lll} <operand_1><operand_2><operator> & \rightarrow & \text{LDA} \quad <operand_1> \\ & & <op> \quad <operand_2> \\ & & \text{ST} \quad <temp> \end{array}$$

where the $<op>$ in the resultant set of strings is dependent on the $<operator>$ in the source string. In particular, the following correspondences will be in effect:

Input String Operator	*Output String Operator*
+	ADD
−	SUB
*	MUL
/	DIV

In the particular case of the replacement sign, the production strings are

$$\begin{array}{lll} <operand_1><operand_2> = & \rightarrow & \text{LDA} \quad <operand_2> \\ & & \text{ST} \quad <operand_1> \end{array}$$

In these productions, $<temp>$ refers to an infinite set of memory addresses which are free of other uses. It is assumed that once an element has been used in this set it is no longer available for use; that is, it is no longer free.

From Polish to Machine Code

Let us review the rules for converting Reverse Polish notation to normal form:

1. Scan from left to right to locate the first operator.

2. The two preceding operands are then the operands of that operator.

3. Exchange the positions of the operand immediately to the left of the operator and the operator, at the same time enclosing the operands and the operator in parentheses. For all further processing this parenthesized group is regarded as an operand or, when enclosed in further parentheses, as part of an operand.

4. If the last parentheses added comprise a pair that encompasses the whole string, then the process is complete. If not, then continue to scan to the right until another operator is located, and return to step 2.

This same process can be used to generate a set of machine instructions if step 3 is changed to the following:

3. Use the production (on page 182) to generate instructions appropriate to the operands and operator. Replace the three elements of the input string by the name of the temporary storage location.

Using this new set of rules, let us consider the input string

$$xabc/{*}d+=$$

At the first scan, the located operator is / which has the operands b and c. Thus using the production rule appropriate to /, we shall generate the instructions

$$\begin{array}{ll}\text{LDA} & b\\ \text{DIV} & c\\ \text{ST} & \tau_1\end{array}$$

where τ_1 is a free element from the set $<temp>$. The string then is reduced to $xa\tau_1{*}d+=$. Continuing to scan to the right, the next operator is * and its operands are a and τ_1; hence the generated instructions are

$$\begin{array}{ll}\text{LDA} & a\\ \text{MUL} & \tau_1\\ \text{ST} & \tau_2\end{array}$$

and the string is reduced to $x\tau_2d+=$. Further applications of the rules of generation will producé the instructions:

LDA τ_2
ADD d
ST τ_3
LDA τ_3
ST x

It is obvious that in the total set of generated instructions there are, in certain instances, back to back store and load instructions that reference the same operand. That is, a result is stored in a temporary location by the use of one production and is brought back immediately by the next production. Further, in those cases where the operator is commutative, a temporary storage location can be saved by reversing the positions of the operands. However, although these deficiencies may be recognized at this time, we shall postpone detailed consideration of them until a later section.

From Algebraic Notation to Machine Code

Machine instructions may be generated from the normal algebraic string if the algorithm above is preceded by the instructions describing conversion from normal form through parenthesized form to Reverse Polish notation. This technique, which culminates the considerations of the previous chapter together with the above algorithm, may be described as follows:

1. Assign hierarchy levels to each operator taking into account the step increases and decreases when a parenthesis is encountered.

2. Starting from the left-hand end of the statement, locate the operator such that $V(op_n) \geq V(op_{n+1})$.

3. The operands immediately to the left and right of this operator are the operands pertinent to the chosen operator.

4. (Note that at this point in the algorithm to convert to Reverse Polish notation we would have interchanged the right-hand operand and the operaor and enclosed the two operands and the operator in braces. In this algorithm we merely rewrite the productions.) Using the following productions, generate the appropriate instructions:

$$\langle operand_1\rangle\langle operator\rangle\langle operand_2\rangle \quad \rightarrow \quad \begin{array}{ll} \mathsf{LDA} & \langle operand_1\rangle \\ \langle op\rangle & \langle operand_2\rangle \\ \mathsf{ST} & \langle temp\rangle \end{array}$$

except where the operator is the replacement sign in which case the production rule is

$$\langle operand_1\rangle = \langle operand_2\rangle \quad \rightarrow \quad \begin{array}{ll} \mathsf{LDA} & \langle operand_2\rangle \\ \mathsf{ST} & \langle operand_1\rangle \end{array}$$

5. Replace the operator and its operands by the name of the temporary location in which the result is being stored.

6. If the operator that forced the compilation of the instructions pertinent to op_{n} is the end of the statement ($\vdash$), return to step 2, unless the statement is now null, which indicates that the compilation (or generation) is complete.

7. If the forcing operator is not the end of the statement, check $\mathsf{op}_{\mathsf{n}-1}$ for the possibility that the forcing operator will also force the compilation of this operator. That is, set the pointer to the operator previous to the one just compiled and return to step 2 without returning the pointer to the left-hand end of the statement.

Let us consider an input string in normal form which is to be compiled by this set of instructions. In the following discussion, the symbols

$$\overset{\uparrow}{\mathsf{C}} \text{ and } \overset{\uparrow}{\mathsf{H}}$$

are used to indicate the operators that are *c*urrently being investigated and that which is in *h*and as a possible forcing operator. Given

$$\begin{array}{ccccccccccc} \vdash & x & = & a & * & b & / & c & + & d & \dashv \\ 0 & & 1 & & 3 & & 3 & & 2 & & 0 \end{array}$$

the first scan will place the current operator pointer under the * symbol while that in hand will be the division operator. That is,

$$\begin{array}{ccccccccccc} \vdash & x & = & a & * & b & / & c & + & d & \dashv \\ & & & & \uparrow & & \uparrow & & & & \\ & & & & \mathsf{C} & & \mathsf{H} & & & & \end{array}$$

At this point, the compilation of the multiply operator and its operands is forced, thus producing the instructions

$$\begin{array}{ll} \mathsf{LDA} & a \\ \mathsf{MUL} & b \\ \mathsf{ST} & \tau_1 \end{array}$$

and the string is reduced to

$$\begin{array}{ccccccccc} \vdash & x & = & \tau_1 & / & c & + & d & \dashv \\ & & \uparrow & & \uparrow & & & & \\ & & \mathsf{C} & & \mathsf{H} & & & & \end{array}$$

which shows that the in hand pointer remains in position and the current operator pointer moves to the left. At this point, the hierarchial level of the in hand operator does not force the compilation relevant to the current operator, and so we will scan to the right, until the following situation exists:

$$\begin{array}{ccccccccc} \vdash & x & = & \tau_1 & / & c & + & d & \dashv \\ & & & & \uparrow & & \uparrow & & \\ & & & & \mathsf{C} & & \mathsf{H} & & \end{array}$$

The divide operator is now forced to be compiled, and because of the reduction of the string, its operands are still immediately adjacent to the operator. Thus the instructions

$$\begin{array}{ll} \mathsf{LDA} & \tau_1 \\ \mathsf{DIV} & c \\ \mathsf{ST} & \tau_2 \end{array}$$

are generated, and the string is reduced to

$$\begin{array}{ccccccc} \vdash & x & = & \tau_2 & + & d & \dashv \\ & & \uparrow & & \uparrow & & \\ & & \mathsf{C} & & \mathsf{H} & & \end{array}$$

which does not cause the compilation of the replacement operator. At the next shift of the pointers, the addition is forced:

$$\begin{array}{ll} \mathsf{LDA} & \tau_2 \\ \mathsf{ADD} & d \\ \mathsf{ST} & \tau_3 \end{array}$$

The string is then reduced to

$$\begin{array}{ccccc} \vdash & x & = & \tau_3 & \dashv \\ & & \uparrow & & \uparrow \\ & & \mathsf{C} & & \mathsf{H} \end{array}$$

at which point the replacement operator is compiled:

```
LDA   τ3
ST    x
```

and the string is reduced to

$$\vdash \quad \dashv$$

which is obviously empty, thereby signifying that the compilation is complete.

This discussion does not include provisions for the compilation of either the involution operator or the unary minus. The former has been omitted because most computers do not include a single instruction for the computation of an involute. For the purposes of discussion, let us assume that there exists, at least in the assembly language, an instruction of the form

```
EXP   a
```

which means $ACC^a \rightarrow ACC$. This may, in fact, assemble to a set of instructions linking to a routine that will execute this operation. The unary minus is a special case of a function that requires a single argument but does not necessarily compile as a linkage to a routine to compute the function. For the purposes of this text, assume that the computer contains an instruction to reverse the sign of the accumulator. This instruction will not require an operand explicitly since it is implied that the operand is the accumulator. Thus the production for unary minus will be

```
~<operand>  →  LDA    <operand>
               RVSG
               ST     <temp>
```

Let us now return to the problem of using temporary accumulators and the redundancy of store and load instructions that use the same operand in sequence. The accumulator may be regarded as a temporary storage location until its use is precluded by the necessity to store more than one intermediate result or until its position as an operand in the reduced string will not permit the compilation of the instructions as defined above. In other words, so long as the accumulator is the first operand of a binary operator or the only operand of a function, a temporary storage location is not needed. Further, instructions in the above productions either to store into or load out of the accumulator will be omitted except in the case of the replacement sign or when the accumulator is empty.

Consider the following input string:

$$\begin{array}{ccccccccccc} \vdash & x & = & a & * & b & + & c & * & d & \dashv \\ 0 & & 1 & & 3 & & 2 & & 3 & & 0 \end{array}$$

During the first scan from left to right, when the first compilation is forced, the pointers are set to the following positions:

$$\begin{array}{ccccccccccc} \vdash & x & = & a & * & b & + & c & * & d & \dashv \\ & & & & \uparrow & & \uparrow & & & & \\ & & & & \mathsf{C} & & \mathsf{H} & & & & \end{array}$$

Under these conditions the coding

```
LDA   a
MUL   b
```

is generated; the result is stored in the accumulator, and the store instruction to a temporary location is not output. The reduced string becomes

$$\begin{array}{ccccccccc} \vdash & x & = & \mathsf{ACC} & + & c & * & d & \dashv \\ & & \uparrow & & & & \uparrow & & \\ & & \mathsf{C} & & & & \mathsf{H} & & \end{array}$$

In this situation, the coding does not have to be generated for the replacement sign, and thus the scan from left to right is continued. However, the movement of the pointers does not immediately cause the compilation of the operator with which the accumulator is associated. In fact, the next operator to be forced into a compilation is the succeeding multiply operator. Therefore since this does not involve the result that was saved in the accumulator, the result must be saved in a temporary location to allow the use of the accumulator in the next unassociated production.

Thus we may state the following rule: If the in hand pointer is moved and the previous in hand operator does not become the operator for the next production, the result of any previous calculation must be stored temporarily and that storage location must be substituted for the accumulator in the reduced input string.

Consider the example:

$$x = {\sim}a * b + c$$

The Reduced String and Decisions	*Generated Instructions*		
$\vdash x = {}^{\sim}a * b + c \dashv$ 0 1 5 3 2 0 ↑ ↑ C H	LDA	a	$a \rightarrow$ ACC
	RVSG		$-$ACC $\rightarrow$ ACC
$\vdash x =$ ACC $* \ b + c \dashv$ ↑ ↑ C H V(=) $\leqslant$ V(*); move right. $\vdash x =$ACC $* \ b + c \dashv$ ↑ ↑ C H The previous in hand operator is now to be compiled, and the ACC is the left-hand operand; thus no temporary storage is needed.	MUL	b	ACC$*b \rightarrow$ ACC
$\vdash x =$ ACC $+ \ c \dashv$ ↑ ↑ C H V(=) $\leqslant$ V(+); move right. $\vdash x =$ ACC $+ \ c \dashv$ ↑ ↑ C H The previous in hand operator is now to be forced; no temporary storage is needed.	ADD	c	ACC$+c \rightarrow$ ACC
$\vdash x =$ ACC $\dashv$ ↑ ↑ C H At this point, the in hand operator is not the next operator to be compiled. However, the replacement operator is to be forced and the accumulator is in the correct relationship with the operator and is to compile normally	ST	x	ACC $\rightarrow x$
$\vdash \ \dashv$ The string is now empty; hence the compilation is complete.			

Now consider the string

$$x = a + b \,/\, c \uparrow d - e$$

The Reduced String and Decisions — *Generated Instructions*

$$\begin{array}{ccccccccccccc} \vdash & x & = & a & + & b & / & c & \uparrow & d & - & e & \dashv \\ 0 & & 1 & & 2 & & 3 & & 4 & & 2 & & 0 \\ &&&&&&&& \uparrow && \uparrow && \\ &&&&&&&& \mathsf{C} && \mathsf{H} && \end{array}$$

LDA	c	$c \rightarrow$ ACC
EXP	d	ACC$\uparrow d \rightarrow$ ACC

$$\begin{array}{ccccccccccc} \vdash & x & = & a & + & b & / & \mathsf{ACC} & - & e & \dashv \\ &&&&&& \uparrow && \uparrow && \\ &&&&&& \mathsf{C} && \mathsf{H} && \end{array}$$

Immediately after the compilation of the operator of involution, the pointers are set as shown above. In this situation, the in hand operator did not become that to be compiled next. Further, although the forced operator (/) does have the accumulator as one of its operands, it is the divisor that is in the accumulator and not the dividend. Hence the contents of the accumulator must be stored temporarily.

ST	τ_1	

$$\begin{array}{ccccccccccc} \vdash & x & = & a & + & b & / & \tau_1 & - & e & \dashv \\ &&&&&& \uparrow && \uparrow && \\ &&&&&& \mathsf{C} && \mathsf{H} && \end{array}$$

Having stored the contents of the accumulator, we are effectively compiling a new statement. Thus the first operation will be that of loading the accumulator.

LDA	b	$b \rightarrow$ ACC
DIV	τ_1	ACC/ $\tau_1 \rightarrow$ ACC

$$\begin{array}{ccccccccc} \vdash & x & = & a & + & \mathsf{ACC} & - & e & \dashv \\ &&&& \uparrow && \uparrow && \\ &&&& \mathsf{C} && \mathsf{H} && \end{array}$$

The Reduced String and Decisions	*Generated Instructions*		
Once again the forced operator is not that which was in hand, and therefore the contents of the accumulator must be saved.	ST	τ_2	ACC → τ_2
$\vdash x = a \underset{\substack{\uparrow \\ \text{C}}}{+} \tau_2 \underset{\substack{\uparrow \\ \text{H}}}{-} e \dashv$	LDA ADD	a τ_2	a → ACC ACC+ τ_2 → ACC
$\vdash x \underset{\substack{\uparrow \\ \text{C}}}{=} \text{ACC} \underset{\substack{\uparrow \\ \text{H}}}{-} e \dashv$			
After the generation of the instructions pertinent to the addition operator, the pointers are set as shown above, and since the V(=) is less than V(−), then the previous operator is not forced. A single shift left of the pointers will indicate that the in hand operator becomes that to be compiled. Thus no temporary storage location is needed.	SUB	e	ACC − e → ACC
$\vdash x \underset{\substack{\uparrow \\ \text{C}}}{=} \text{ACC} \underset{\substack{\uparrow \\ \text{H}}}{\dashv}$			
In the above situation, the in hand operator has not become that to be forced; however, the operator being forced is the replacement sign, with the accumulator in the correct relationship.	ST	x	ACC → x
$\underset{\substack{\uparrow \\ \text{C}}}{\vdash} \; \underset{\substack{\uparrow \\ \text{H}}}{\dashv}$			
Compilation is complete.			

In the latter example, two instructions and a storage location could hav been saved if it had been recognized that an add operation is commutative for although the accumulator was not in the correct relationship to th operator with respect to our rules of productions, the add operation coul have been generated by the first operand. In general, the commutativ operations are particular cases which are in a minority among the tota number of operators. However, a survey of operations executed will shov that these two operations (+ and *) are most frequently used. Thus i would be advantageous to arrange the productions to produce efficien codings of these operations. With respect to the subtraction operation, if instead of the in hand operator, the operator immediately to the left of th last forced operator is the one to be compiled, the sequence of instruction RVSG and ADD may be used to simulate the subtraction without the use o a temporary storage location. For example,

$$x = a - b * c$$

```
The Reduced String and Decisions          Generated Instructions
--------------------------------------------------------------------
|– x = a – b * c –|
 0   1   2   3   0
             ↑   ↑
             C   H                  LDA    c     c → ACC
                                    MUL    b     ACC*b → ACC
|– x = a – ACC –|
         ↑     ↑
         C     H

At this point the contents of the
accumulator are to be subtracted
from the value of the variable a.   RVSG         –ACC → ACC
                                    ADD    a     ACC+a → ACC
|– x = ACC –|
   ↑       ↑
   C       H
                                    ST     x     ACC → x
|–  –|
↑   ↑
C   H
Compilation is complete.
```

This saving in instructions and storage cannot be extended to the other noncommutative operators / and ↑. However, there can be an overall saving within a program if reverse divide and reverse involution are performed by a subroutine. That is, if the subroutine will save the contents of the accumulator in a private (local) storage area, the first operand can be placed in the accumulator and the operation performed in the normal manner; the only instructions needed in the program proper are the linkage.

Suppose n instructions are required to link from the main line program to the subroutine which itself consists of m instructions, whereas in line coding p instructions would be required; then if $p > n$ and the operation is repeated in the total program more than $m/(p-n)$ times, there is a saving in instructions but not necessarily in time. The compiler writer, knowing the environment in which the system is to be used, must determine whether it is more important to save memory space or execution time. For the sake of the present discussion, consider two macro-instructions:

RDIV	a	$a/$ACC → ACC
REXP	a	$a\uparrow$ACC → ACC

For example,

$$x = a - b \;/\; c \uparrow (e - f)$$

Reduced String	*Generated Instructions*
$\vdash x = a - b \;/\; c \uparrow e - f \dashv$ 0 1 2 3 4 12 0 ↑ ↑ C H	LDA e SUB f
$\vdash x = a - b \;/\; c \uparrow$ ACC $\dashv$ ↑ ↑ C H	REXP c
$\vdash x = a - b \;/$ ACC $\dashv$ ↑ ↑ C H	RDIV b
$\vdash x = a -$ ACC $\dashv$ ↑ ↑ C H	RVSG ADD a

Continued on p. 194.

Reduced String	*Generated Instructions*
$\vdash x \underset{\underset{\text{C}}{\uparrow}}{=} \text{ACC} \underset{\underset{\text{H}}{\uparrow}}{\dashv}$	ST x
$\underset{\underset{\text{C}}{\uparrow}}{\vdash} \quad \underset{\underset{\text{H}}{\uparrow}}{\dashv}$	

Compilation is complete.

Problem

8.1 Show the successive steps of analysis to code the following arithmetic statements:

(a) $x = (-b + (b * b - f * a * c) \uparrow h) \,/\, (t * a)$

(b) $y = (-a * b + c) \uparrow (-(d - e))$

(c) $z = -d - a + b \,/\, (c * e - f \,/\, g)$

(d) $v = (a + b) - (c + d)$

(e) $n = -a \uparrow b \,/\, (-(a \uparrow b))$

In normal practice, the hierarchy of operators places a unary minus at a higher level than all other operators; thus expressions such as $a * -b$ and $a \uparrow -b$ are parenthesized so that the sign of the second operand is always reversed before the operation to its left is executed. However, this also implies that the following translations:

$$-a * b \;\rightarrow\; ((-a) * b)$$
$$-a \uparrow b \;\rightarrow\; ((-a) \uparrow b)$$

The latter is a possible meaning, but, in general, if b is a mixed number (integer + fraction), the result of $((-a) \uparrow b)$ is undefined. Therefore let us define $-a \uparrow$ b as meaning $-(a \uparrow b)$. While $-(a * b)$ can be accepted as a meaning for $-a * b$, $-a + b$ cannot be translated as $-(a + b)$. Thus if the unary minus is placed between $+,-$ and $*,/$ the resultant meaning will be as desired. However, this meaning creates some ambiguity as to the possibility of the occurrence of two operators without separating operands. To overcome this problem, let us insist that no two operators may be placed in sequence except when the replacement sign and unary minus occur in juxtaposition. Thus the hierarchial table is revised to the sequence

0	⊢	⊣
1	=	
2	+	− (binary)
3	− (unary)	
4	*	/
5	↑	

with the adjustments of +10 for (and −10 for).

If the reverse operations for subtraction and division and involution are regarded as standard operators, scanning from the left after the initial location of a prime operator is simpler and conserves time since, in general, the majority of the scanning is to be performed to the left. Consequently, as a regular procedure, the operand to the left of the in hand operator is placed in the accumulator first. Consider the input string $x = a - b$.

```
       Reduced String                   Generated Instructions
--------------------------------------------------------------
⊢ x = a − b ⊣
0   1   2   0
        ↑   ↑
        C   H                           LDA    b
                                        RVSG
                                        ADD    a

⊢ x = ACC ⊣
    ↑     ↑
    C     H                             ST     x
```

In this particular production sequence, a two instruction sequence, not a true reverse subtract operation, has been introduced. Another technique of simulation for the reverse subtract is:

```
SUB   a     ACC − a → ACC
RVSG        −ACC → ACC
```

Let us consider the previous example:

```
        Reduced String                  Generated Instructions
---------------------------------------------------------------
⊢ x = a − b ⊣
0   1   2   0
        ↑   ↑
        C   H                          LDA    b
                                       SUB    a
                                       RVSG
⊢ x = ACC ⊣
    ↑     ↑
    C     H                            ST     x
```

If the expression being compiled had been

$$x = -(a - b)$$

the instructions generated would have been as follows:

```
        Reduced String                  Generated Instructions
---------------------------------------------------------------
⊢ x = − ( a − b ) ⊣
0   1 3       12    0
            ↑     ↑
            C     H                    LDA    b
                                       SUB    a
                                       RVSG
⊢ x = − ACC ⊣
      ↑     ↑
      C     H                          RVSG
⊢ x = ACC ⊣
    ↑     ↑
    C     H                            ST     x
```

In this case, the accumulator is reversed in sign in two successive instructions, each canceling the effect of the other and therefore being redundant. This effect was discussed in Chapter 7 with respect to moving reverse sign operations (unary minuses) through an expression in order to find a canceling operation. During compilation, the order in which instructions are generated is controlled somewhat by the various hierarchial levels of the operators, and thus complete analysis of the expression, as was performed in Chapter 7, is not necessarily economical. However, if the instructions are ordered so that the reverse sign operation is the last to be generated in a production, then there is a chance of either canceling it with a subsequent reverse sign operation or combining it with another operator. The algorithm of generation should take note of the generation of a reverse sign operation and check to see if the next instruction is also a reverse sign. If this is the case, then both the generated instructions may be canceled before they are passed into the object code. Further, if the operator to be generated next is either an ADD or SUB and the accumulator is the right-hand operand, then these operations may be converted to their opposite operation without affecting the true representation of the expression. Consider the following examples.

Reduced String	*Generated String*	
	Without Reversal	*With Reversal*
$\vdash x = a - (b - c) \dashv$ 0 1 2 12 0 ↑ ↑ C H	LDA c SUB b RVSG	LDA c SUB b
$\vdash x = a - \text{ACC} \dashv$ ↑ ↑ C H	SUB a RVSG	ADD a
$\vdash x = \text{ACC} \dashv$ ↑ ↑ C H	ST x	ST x
$\vdash x = a + (b - c) \dashv$ 0 1 2 12 0 ↑ ↑ C H	LDA c SUB b RVSG	LDA c SUB b
$\vdash x = a + \text{ACC} \dashv$ ↑ ↑ C H	ADD a	SUB a RVSG
$\vdash x = \text{ACC} \dashv$ ↑ ↑ C H	ST x	ST x

The instructions generated in the first example show that there is a saving of two instructions when reversal is taken into account, whereas in the second example, there is no saving. The strings generated with reversal show that when the reverse sign instruction is kept back until the next productions are generated, the reversal of the ADD or SUB instructions can save instructions and, if not, will not be detrimental to the program. It is better to save instructions occasionally, than to allow the generation of uneconomical sets of instructions.

In each of the above examples, the in hand operator has forced the compilation of the left-hand operator. When a move to the right is needed, the production rules are different, since in this case the accumulator is the left-hand operand and the normal mode of operations is possible. For example, the input string $x = a - b - c$ produces the instructions

```
LDA   b
SUB   a
RVSG
SUB   c
ST    x
```

However, if the subsequent subtract had been converted to an add while the reverse sign was maintained on the outside of the grouping, the same number of instructions would have accrued. That is,

$$(x=a-b-c) \equiv (x=-(b-a)-c) \equiv (x=-((b-a)+c))$$

the latter generating the coding

```
LDA   b
SUB   a
ADD   c
RVSG
ST    x
```

This allowance in the algorithm does not show any increase in the object coding, but the movement of the unary minus (reverse sign operation) to the outside of the expressions and subexpressions increases the possibility of the unary minus being canceled.

If a unary minus is located or brought to the outside of a group and the next operator is either a * or a / without an intermediate store in a temporary location, the unary minus may be carried further. However, if the next operator is the involution operator, the RVSG must be executed independently of the direction of the scan.

Problem

8.2 Generate the coding for the following expressions, indicating the process of production step by step.

(a) $-\left[\frac{a}{-b}+(-c)\right]$

(b) $(-ab+c)^{-(d-e)}$

(c) $-(a-(c*(-b/(d^{-k}))))$

Scanning by Tables

Within a computer, it is not economical to analyze an arithmetic expression in its algebraic linearized form. It is particularly inconvenient to keep moving about an expression and replacing groups by a notation which indicates that the result is either in the accumulator or in temporary storage. Further, as the expression is analyzed, it becomes vacuous, and time is wasted if the compiler must scan blanks in the statement and blocks that have already been scanned, or must condense the statement.

To avoid these problems, let us scan the statement left to right and place the operands in a table of encountered addresses (TEA) and the operators in a table of encountered operators (TEO). For this illustration, we shall also maintain two other tables: one of operator levels and one containing the location of the in-hand pointers. Even before each statement is examined, the left-hand end delimiter ($\vdash$) is placed in TEO, and when the end is located,† the right-hand end delimiter ($\dashv$) is generated. As the scan is conducted, each new operator encountered (except the end delimiter to the left of the statement) is placed in the "in-hand" bin. If the new operator is of a lower level or equal level to the operator last placed in TEO, then the last two operands are used and the last operator placed in TEO is used in the production that generates the required code. These operands and operator are removed from their respective table, and a new operand is placed in TEA, indicating that the result is in the accumulator. The "in-hand" operator is then compared to the (new) last operator in TEO. If this operator is still of a higher hierarchical value than that in hand, this operator is also

† The ease with which the end of an arithmetic expression is located depends on the statement in which it is embedded. For example, if the statement under consideration is an assignment statement, then the end of the expression is also the end of the statement. In other types of statement (for example, the IF statement) the decision-making machinery for recognizing the end of the expression may be more complex.

forced into the compilation sequence. If not, the scan is continued to the right, and the in-hand operator and operand are placed in TEO and TEA, respectively. However, if two operands † are added to TEA before another compilation is forced, then the previous result (now residing in the accumulator) must be stored in a temporary storage location, and that location must be recorded in the correct position in TEA. Consider the input string

$$x = a - b * c$$

TEA	TEO	*Level*	*In Hand*
x	⊢	0	
a	=	1	
b	−	2	
c	*	3	⊣

At this point in the construction of the table, V(⊣) $\leqq$ V(*) and the accumulator does not contain anything. Hence, the last entry in TEA is placed in the accumulator and the entry removed from TEA:

LDA *c*

The last operator in TEO is *, and its operand is *b:*

MUL *b*

The last two entries are removed from TEA and TEO, respectively, and the table reduces to:

TEA	TEO	*Level*	*In Hand*
x	⊢	0	
a	=	1	
	−	2	⊣

At this point, V(⊣) is still less than V(last entry in TEO), so we are, in effect, scanning left. The accumulator is presently in use so it does not have to be loaded. Thus the last operator and operand are the relevant items for the construction of the next instruction:

SUB *a*

† For the time being, only binary operators are being considered.

The next instruction to be generated should be the RVSG, which will be held in reserve until the items relevant to the last instruction generated are removed from the tables. The reduced table now shows that the replacement sign is the last entry in TEO, and the in hand operator is still the right-hand end delimiter. Since no more items are to be added to the table and the in hand operator is forcing the compilation of the instructions pertinent to the replacement sign, the RVSG instruction will not have the opportunity to interact with any other instructions. Thus it may be placed into the object code. The table is now:

TEA	TEO	*Level*	*In Hand*
x	$\vdash$	0	
	$=$	1	$\dashv$

and since V(in hand operator) is still less than V(last entry in TEO), the instruction

```
ST   x
```

is forced. The TEA table is now empty, and TEO contains only the left-hand delimiter which is to be matched with the right-hand delimiter in the in hand column. Thus the compilation is complete.

Now let us consider the expression

$$x = a * b + c * d$$

Scanning from left to right, the tables are built up until the following situation exists:

TEA	TEO	*Level*	*In Hand*
x	$\vdash$	0	
a	$=$	1	
b	$*$	3	$+$

At this point, V(in hand operator) is less than V (*), and the accumulator is empty since no coding has been produced previously. Hence the generator produces the following instructions:

```
LDA   b

MUL   a
```

and the tables reduce to:

TEA	*TEO*	*Level*	*In Hand*
x	⊢	0	
	=	1	+

But, V(in hand operator) is not less than or equal to V(TEO operator), and so the ACC must be added to the TEA and the scan continued.

TEA	*TEO*	*Level*	*In Hand*
x	⊢	0	
ACC	=	1	
c	+	2	*

At this point, the in hand operator of the last production is not forced into the compilation cycle by the new in hand operator. Therefore the contents of the accumulator must be stored in a temporary storage location and the TEA updated to reflect this storage. The instruction produced is

```
ST    τ1
```

Scanning of the input statement is continued and the table is constructed to the point, shown below, where the level value of the in hand operator forces the next compilation:

TEA	*TEO*	*Level*	*In Hand*
τ_1			
x	⊢	0	
τ_1	=	1	
c	+	2	
d	*	3	⊣

Now the accumulator may be considered empty since the last computed result has been placed in temporary storage. Thus the generated instructions are

```
LDA   d
MUL   c
```

After the last operator and its operands are removed from the tables, the level value of the in hand operator is still less than that of the last entry in TEO, thus forcing the compilation of this operator:

ADD τ_1

Since the end delimiter always has a level value less than every other operator, except the left-hand delimiter, all operators will be forced into the compilation cycle. In this case, only one other operator exists in the TEO besides the left-hand delimiter, that is, the replacement sign. This causes production of the instruction

ST x

after which the tables are empty and the compilation is complete.

Let us now consider an example involving a reversal of an operation caused by the interaction of a unary minus (or RVSG): The input string $x = a - (b - c)$ contains parentheses that will be discarded during the scan and will not be placed in TEO. However, when the left parenthesis is encountered, all levels of hierarchy are incremented by 10, and when the closing parenthesis is recognized, the levels are reduced by 10. If at any point the levels become negative, then there are additional closing parentheses that appear before their matching opening parentheses, and an illegal statement is in the input area. Similarly, if the level of the end delimiter is not zero after a series of hierarchial incrementations and decrementations, then unmatched parentheses exist in the input string.

For the above string, the table is constructed to the point where the end delimiter is the in hand operator.

TEA	*TEO*	*Level*	*In Hand*
x	⊢	0	
a	=	1	
b	−	2	
c	−	12	⊣

Since no instructions have been generated up to this point, the accumulator is not in use, and hence the first instruction must be to load the accumulator with the last entry in TEA.

LDA c
SUB b

The RVSG instruction is held in reserve; the tables are reduced by the operands and the operator used in the last production. The last entry in TEO is now −, which is to be forced by the in hand operator. In effect, the scan is proceeding to the left across the input string; therefore the unary minus (RVSG) will interact with this operator and convert it to an ADD operator, and the RVSG instruction which is being held in reserve will be discarded. The table is thus reduced and modified to

TEA	*TEO*	*Level*	*In Hand*
x	⊢	0	
a	=	1	
	+	2	⊣

The in hand operator, which is the end delimiter, will now force the compilation of the instructions relevant to the last entry in TEO, and since the accumulator is still in use, no temporary storage is required. The instruction generated is ADD a. Finally, the in hand operator forces the compilation of the replacement operator, ST x.

Now, consider the string $x = a - b - c$.

TEA	*TEO*	*Level*	*In Hand*
x	⊢	0	
a	=	1	
b	−	2	−

At this point, the accumulator is not in use, and compilation of the operator, −, is being forced by an operator of the same level.

```
LDA  b
SUB  a
```

After the operands and the operator have been removed from the tables (and an RVSG is pending), the current operator (that is, the item in TEO that is now unmasked) will not be forced by the in hand operator and thus the table must be constructed further until the next forced operator is discovered. The pending RVSG operation is equivalent to a unary minus that is operating on the accumulator and will be placed in the TEO. Normally, this should be forced into compilation except under one condition: if the next operand (which is currently in hand) is to be forced at the next stage.

If the next operand is not to be forced then this operation must be forced. In this example, the in hand operator is the next to be forced and will be converted to +. Thus before the next stage of compilation, the tables are as follows:

TEA	TEO	*Level*	*In Hand*
x	⊢	0	
ACC	=	1	
c	$\sim$	3	
	+	2	⊣

In effect,

$$\sim\text{ACC} - c$$

has been converted to the equivalent form $\sim(\text{ACC} + c)$. If the next operator had not been forced, then the reverse sign operation would have had to have been compiled into the object coding. However, if for any reason, the next operator had been either a multiply or divide, the reverse sign could have remained in the table. In the above case, the compiled instructions are, successively:

```
ADD   c
RVSG
ST    x
```

Problem

8.3 Using the tabular method of compilation, generate the instructions for the following input strings:

(a) $x = \sim a - b - c - d$

(b) $x = \sim(a-b) + (c - d)$

(c) $x = \dfrac{\sim a}{cb} - e$

(d) $x = \dfrac{\sim b + \sqrt{b^2 - 4ac}}{2a}$

(e) $y = (x_i + x_{i+1})/z_{n-2}$

(f) $y = \cos(x)$

Subscripted Variables[†]

Before considering the actual inclusion of subscripted variables in an arithmetic statement which is to be compiled, two points should be noted:

(a) Subscripted variables are not peculiar in an arithmetic statement. They play the same part as any simple variable. In other words, subscripted variables vary from simple variables in their notation but not in their use.

(b) The computation required to locate the address of a subscripted variable (or in purer mathematical terms, to locate the element of an array), is dependent on the dimensions of the whole array in which the element is located.

When a DIMENSION statement is located in a program, not only is space reserved for that array but also information is generated which aids the compilation of instructions to generate the address of the particular element under consideration. As a review of this procedure (described in Chapter 4), consider a two-dimensioned array specified in the statement:

DIMENSION A(5,3)

and a reference in an executable statement to the element A(2,2). At the time the storage for A is set up in SYMTAB, the address of the base of A (that is, the fictitious element A(0,0)) is computed and is available to the compiler at the time when the instructions to locate the particular element are generated. The location of A(2,2) may be computed from the expression

$$\text{Address}(\text{A}(2,2)) = \text{Address}(\text{Base of A}) - (2*5 + 2)$$

In general, if one is presented with a DIMENSION statement containing the specification

$$\text{DIMENSION A}(d_1\,, d_2\,, d_3\,, \ldots d_N)$$

[†] In this discussion, it is assumed that the object time memory layout is such that all data are stored in high order memory, with elements of arrays placed in descending locations. Such a practice has not been followed in IBM System/360, where arrays are stored in ascending storage locations. See IBM form No. C28-6515-4, IBM System/360, FORTRAN IV Language.

Further, the formulas must be amended when either the data being referenced are stored in multiple word blocks (for double precision or complex data) or the computer on which the compiler is being implemented is character rather than word oriented.

and wishes to refer to any element

$$\mathsf{A}(s_1\ , s_2\ , s_3\ , \ldots s_N)$$

then one can evaluate the expression

$$\begin{aligned}&\mathsf{Address}(\mathsf{A}(1,1,1,\ldots 1)) - (s_1 -1) - d_1\,(s_2 -1) - d_1d_2\,(s_3 -1)\\ &\qquad - d_1d_2d_3\,(s_4 -1) - \ldots - d_1d_2d_3 \ldots d_{N-1}\,(s_N -1)\end{aligned}$$

That is,

$$\mathsf{Address}(\mathsf{A}(s_1\ ,s_2\ ,s_3\ , \ldots s_N)) = \mathsf{Address}(\mathsf{A}(1,1,1,\ldots 1)) - \sum_{i=1}^{N} \{ \prod_{k=0}^{i-1} d_k\} (s_i -1)$$

where it is assumed that $d_0 = 1$. Since the coefficients of the product will be required each time a subscripted variable address is to be calculated, one should calculate this product when the DIMENSION statement is encountered, rather than leave it for the individual generators to calculate.

If

$$p_i = \prod_{k=0}^{i-1} d_k \qquad \text{and} \qquad p_1 = 1$$

then

$$p_{i+1} = d_i p_i, \qquad \text{for} \qquad 1 < i \lesseqgtr (N-1)$$

and hence

$$\text{Address of element} = \mathsf{Address}(\mathsf{A}(1,1,1,\ldots 1)) - \sum_{i=1}^{N} p_i\ (s_i -1)$$

That is,

$$\text{Address of element} = \mathsf{Address}(\mathsf{A}(1,1,1,\ldots 1)) + \sum_{i=1}^{N} p_i - \sum_{i=1}^{N} p_i\, s_i$$

Now since the terms

$$\mathsf{Address}(\mathsf{A}(1,1,1,\ldots 1)) \qquad \text{and} \qquad \sum_{i=1}^{N} p_i$$

are constants, they may be combined into a single constant term which is computable at the time that the DIMENSION statement is under consideration. This is the constant that was referred to as the BASE of the array in Chapter 4. In fact, the address of the BASE of an array is also the address of the fictitious element A(0,0,0, . . . 0). If the subscript expressions are restricted to the standard form, that is,

$$<\textit{integer constant}>*<\textit{integer variable}>\{+|-\}<\textit{integer constant}>$$

then all the computations required to compute the address of any element may be performed with the use of only one temporary storage location. In particular, if the computer being used at object time has index registers, the result of the computation of

$$-\sum_{i=1}^{N} p_i$$

may be placed in an available index register, and then the instruction that references that particular element will have the operand address of the base of the array that is influenced by that index register.

In general, a subscript, irrespective of its form, that is standard (c*v±k) or nonstandard (any expression that results in an integer value), may be broken down to the form:

$$<\textit{variable expression}> \pm <\textit{constant expression}>$$

where the <*variable expression*> contains not only variables but also any associated multipliers or divisors that are constants. That is, the subscript

$$(3*\mathrm{I} + 32/\mathrm{J})*4 + 32$$

contains the <*variable expression*>

$$(3*\mathrm{I} + 32/\mathrm{J})*4$$

and the <*constant expression*> +32.

If all subscripts are broken down in this manner, the address of a subscripted variable may be computed from

$$\text{Address(base of array)} - \sum_{i=1}^{N} p_i\, c_i - \sum_{i=1}^{N} p_i\, v_i$$

where c_i is the <*constant expression*> of the subscript s_i, and v_i is the <*variable expression*> of the subscript s_i. At compile time, in the subscript evaluation generator, the base of the array may be further adjusted by the term

$$-\sum_{i=1}^{N} p_i\, c_i$$

thus saving valuable execution time.

During the compilation of a subscript evaluation instruction set, the delimiting commas and parentheses have special meanings and, in fact, imply arithmetic operations. In particular, after the opening parenthesis, the address of the base of the referenced array is to be placed into a special compile time word for further manipulation. If one considers only standard subscripts, it can be seen that the first subscript may easily be separated into its two parts: The instructions to compute the <*variable expression*> may be placed into the object time coding, and then the value of the <*constant expression*> may be evaluated and the result applied to the base address of the array to produce an adjusted base for modification at object time. If this first subscript is also the only subscript needed to refer to an element of the array, instructions to store the resultant element address in a chosen index register may be generated. In fact, this set of instructions will be generated whenever a closing parenthesis of a subscripting set is encountered. On locating any delimiting comma or the closing parenthesis, instructions to multiply the value of the variable portion of the subscript by the p_i value must be generated. To enable the performance of this operation, the constants p_1 to p_N must be stored in the object time data table. At the same time, the value of the constant portion of the subscript must be applied, along with the associated multiplying factor, to the adjusted base.

In the compilation of any general expression, the problem of deciding when to compute the addresses of references to elements of an array cannot be solved, either simply or uniquely. Whenever a subscript evaluation is required, the existing contents of the accumulator must be stored. Consequently, it is not always efficient to attempt to evaluate subscript expressions within the computation of the value of the containing expression. For example, the expression:

```
A(I,J) = B(I,K+3)*C(L) + A(I,J-1)
```

would compile, without subscript evaluation, to the sequence:

```
LDA   C(L)
MUL   B(I,K+3)
ADD   A(I,J-1)
ST    A(I,J)
```

Suppose that the arrays mentioned above are described in the statement:

```
DIMENSION A(5,3), B(10,2), C(25)
```

If the subscript evaluation instructions are superimposed over the above coding, the total set of instructions would be expanded to the set:

```
LIR     L,1
LDA     L
RVSG
LIR     ACC,1
LDA     Base of C,1
ST      τ1
LDA     I
ST      τ2
LDA     K
MUL     C10
ADD     τ2
RVSG
LIR     ACC,2
LDA     τ1
MUL     Adjusted base of B,2
ST      τ2
LDA     I
ST      τ3
LDA     J
MUL     C5
ADD     τ3
RVSG
LIR     ACC,3
LDA     τ3
ADD     Adjusted base of A,3
ST      τ3
LDA     I
ST      τ2
LDA     J
MUL     C5
ADD     τ2
RVSG
LIR     ACC,4
LDA     τ3
ST      Base of A,4
```

where the mnemonic LIR stands for LOAD INDEX REGISTER, and the second operand in the instruction refers to the influencing index register. C5 and C10 are the locations of the constants 5 and 10, respectively.

The above coding suffers from several deficiencies, but two specific ones are worthy of note: (a) If the first subscript has a variable expression that is a single variable, successive LOAD and STORE instructions are generated; (b) as each "main line" operation is performed, the result must be stored in a temporary location. If the subscript evaluations had all been performed first, none of the latter store instructions would have been required. However, the situation would be no better if a complete statement were scanned and all subscript instructions generated before the "main line" scan were executed. Thus in a normal scan, subscript evaluation instructions may be generated as the subscripted variables are encountered. In other words, an opening parenthesis of a subscript should be regarded as a forcing operator; the subscript expression between that parenthesis and the closing parenthesis should be evaluated and then the array name in TEA replaced by the adjusted address and the normal scan continued.

Consider the expression:

$$A(I,J) = B(I,K+3) * C(L) + A(I,J-1)$$

and the tables that are formed to the point:

TEA	*TEO*	*Level*	*In Hand*
A	⊢	0	
I	(	10	,

At this point, a delimiting comma is located within a subscripting group and the accumulator is not in use. Thus one may generate the instructions

```
LDA   I
ST    τ1
```

where the result is placed into a temporary storage location since the forcing operator is a delimiting comma and further subscripting evaluation instructions that will utilize the accumulator are anticipated. Move the subscript and the forcing operator to TEO and continue the scan.

TEA	*TEO*	*Level*	*In Hand*
A	⊢	0	
τ_1	(	10	
J	,	1	)

At this point, the complementary closing parenthesis of the subscripting group is encountered which will force the evaluation of the last subscript:

```
LDA   J
MUL   C5
```

The closing parenthesis will now force the previous subscript and generate the instruction

```
ADD   τ1
```

Next, the opening parenthesis is forced by the closing parenthesis; the sign of the subscript expression must be reversed to effect the correct adjustment of the address, and then the result must be stored in an available index register.[†]

```
RVSG
LIR    ACC,1
```

Subscripting is now complete, and the tables are reduced to the point where an opening and closing parenthesis face each other. These may now be canceled, and the entry for the last item in TEA (which must be the array name) will be replaced by the adjusted address. Since, in this particular instance, the subscripting expressions contained no constants, the adjusted address is the same as the base address of the array, and the scan may continue.

TEA	*TEO*	*Level*	*In Hand*
Adj(A)	⊢	0	
B	=	1	
I	(	10	,

[†] For the purposes of this discussion, assume that the register is capable of holding negative values.

By giving the opening parenthesis of a subscript group the hierarchical level of 10 and a comma the level of 1, the subscript expressions may be forced in a manner close to that of a normal unsubscripted expression. Variations are incurred because the comma is considered an operator, the closing parenthesis has two actions (acts as a comma and an end delimiter), and an in hand closing parenthesis cancels a facing opening parenthesis in TEO while the last entry in TEA is replaced by an adjusted address.

The use of index registers for adjusting base addresses of arrays in order to locate a single element within the array is satisfactory until the computer runs out of index registers. At worst, when one has an expression containing no parenthesized expressions (apart from those defining subscripts) and subscript adjustments are saved until all subscripts have been evaluated, five index registers, at the most, will be required. That is, the worst case would be an expression of the form:

$$A_i \underset{1}{=} B_j \underset{2}{+} C_k \underset{4}{*} D_l \underset{5}{\uparrow} E_m$$

where, according to the forcing techniques of the above system, all adjusted addresses are computed before the total "main line" expression is evaluated. However, when parentheses are included it is possible for an infinite number of registers to be required. For example, the expression

$$A_i = B_j + (C_k + (D_l + (E_m + (F_n + (\ldots$$

would be prepared in the tables in such a manner that each subscript would be evaluated as it occurred with the associated operator being forced and thus denying the use of that index register for other purposes. To overcome this problem, let us evaluate the subscript expression, adjusting the base by the constant part of the subscript, and then store the result in a temporary storage location. The generic entry address in TEA is changed to the address of this temporary location which is to be referenced indirectly. For the purposes of this description, an indirect address will be designated as being negative; that is, $-\tau_n$ will indicate the temporary location τ_n referenced indirectly. In this case, the expression (and corresponding DIMENSION statement)

```
             DIMENSION A(6,3)
A(I+1,K) = SUM + A(I,K-1) * A(I-1,K)
```

compiles to the sequence of instructions:

```
LDA     I
ST      τ1
LDA     K
MUL     C6
ADD     τ1
RVSG
ADD     Base of A − 1
ST      τ1
LDA     I
ST      τ2
LDA     K
MUL     C6
ADD     τ2
RVSG
ADD     Base of A + 1*6
ST      τ2
LDA     I
ST      τ3
LDA     K
MUL     C6
ADD     τ3
RVSG
ADD     Base of A + 1
ST      τ3
```

where the location C6 contains the constant 6. At this point, all operators are forced by the in hand end delimiter and the tables are reduced to

TEA	*TEO*	*Level*	*In Hand*
$-\tau_1$	⊢	0	
SUM	=	1	
$-\tau_2$	+	2	
$-\tau_3$	*	4	⊣

Thus the remainder of the coding generated is

```
LDA     −τ3
MUL     −τ2
ADD     SUM
ST      −τ1
```

For a complete description of the efficient use of index registers and the use of a limited number of registers in the compilation algorithm, see Horwitz et al.[†]

Temporary Storage Locations

The provision of temporary storage locations is truly the task of the symbol table routines but since such provisions are needed by the ASCAN generator, the problem will be considered here. Firstly, the question of how many temporary locations to reserve cannot be answered definitively. Some programs never need a temporary storage location, whereas others need a comparatively large number, especially when such locations are used in the subscripting calculations. Secondly, the compiler implementer must decide whether temporary locations are to be provided from a previously reserved array or from randomly chosen words. Thirdly, the implementer must decide whether or not it is necessary to preserve memory and thus to reuse temporary storage locations as they become free—that is, whether within a single statement a single storage location should be used wherever possible.

Since one cannot determine the exact (or maximum) number of storage locations that will be needed in a program, implementers, who constantly strive for maximum available storage for the person who uses the system, prefer not to set aside a specific section of available memory for temporary storage locations. To reserve memory just in case it has a use seems pointless. Thus the technique of selecting temporary storage locations from an array is discarded in favor of selection of storage as and when it is needed. Further, since FORTRAN and most other algebraic languages consist of a system of discrete statements, as a minimum requirement, temporary storage locations must be recovered before the execution of each statement.

Let us propose the following technique. If a routine requests a temporary storage location, a new location should be chosen by the symbol table routine as if the request were for the storage of a standard variable or constant. However, at the same time, the SYMTAB routine should record this reserval in a special table and note that this location is in use. In the event that there have been previous requests, during the current compilation, for temporary storage locations, this table can be checked for any that are not in use; if available, a previously used location is preferable to the reserval of another memory word.

[†] L. P. Horwitz et al., "Index Register Allocation," *Jour. ACM,* Vol. 13, No. 1, pp. 43–61, Jan., 1966.

Once a temporary storage location has been allocated in a single compilation, it should be reserved for the whole program and reused whenever possible. Thus. all temporary locations are "freed" when the compilation of a single statement is completed. However, ASCAN should also report the freeing of a temporary location during a scan if space is to be conserved. Thus the same location may be used more than once in a single statement.

By fixing the SYMTAB location of the first entry referring to a temporary location, a linked list may be formed for sequential scanning whenever a location is needed or is being freed. If the SYMTAB routine provides the compile time address of the temporary storage location data to the requesting routine, a storage area may be freed without a sequential scan of the whole threaded list. The SYMTAB entry may take the form:

TAG	OBJECT TIME ADDRESS	COMPILE TIME FORWARD LINK	IN USE TAG

The use of temporary storage locations within a subprogram must not conflict with the storage locations being used in the calling program. That is, since a CALL statement may include expressions that involve the use of a temporary storage location for the transference of the value, the subprogram must not, at any point, use these same locations. If all temporary locations are regenerated when compilation of each new statement begins, then the maximum number of temporary accumulators in use by a CALL statement is equal to the number of arguments. Thus if the calling program has used n temporary storage locations, numbered, for example, from 1 to n and the called subprogram requires k arguments, then (provided $k < n$) the subprogram should not use temporary storage locations 1 to k, as those from $k+1$ to n will be available. If the number of temporary storage locations used by the calling program is less than the number of parameters of the subprogram, then a complete new set of locations will be required.

Problem

8.4 Write a FORTRAN program that will compute the numerical value of numerical expressions that contain no parentheses. Each expression will be punched into a data card in the FORMAT(8(A1,F8.3)) where the first alphabetic field may be only a unary operator or a blank. All other alphabetic fields will be binary operators or blank. However, a blank binary operator is to be considered as an end code. No data will be punched in cols. 73–80. Such a data card might take the form

```
+ΔΔΔ5.000*ΔΔ20.000+0003.715/0000.001
↑
Col. 1
```

where Δ signifies a blank column. Since only one column is allocated to an operator and the up arrow is not a standard element of a key punch, the involution operator (**) will be replaced by the alphabetic character E. The program should output the result in the FORMAT(8(A1,F8.3),1H=,E16.8) where the input string is duplicated into the first 72 columns. The program should check for division by zero and other execution errors within its control such as raising a negative value to a fractional power. Note that due to the FORMAT chosen, each piece of data may contain its sign, so that the input string

```
−−0012.000E−1271.972*−1736.000
```

is valid.

Functions in Expressions

When the symbol table routine reports back to the ASCAN routine that it has encountered a FUNCTION, the system to be activated is similar to that which was used in the generation of instructions for the evaluation of the address of an element of an array. However, since the inclusion of a function such as SINF, COSF, etc., requires linking the calling routine to a subprogram which may itself be relocatable, the action of ASCAN depends substantially on the techniques of linkage. If one could be assured that every function had only one argument (even if that argument were an expression), then that argument could be regarded as the right-hand side of an assignment statement, the result being placed in the accumulator. Thus the link could be simply a MARK PLACE & TRANSFER instruction, the subprogram returning control to the main program through a BRANCH BACK and the result being placed in the accumulator.

However, such assurance cannot be given except in the primitive subsets of algebraic languages. In particular, such functions as ATAN(A,B) and MAX(A,B,C,D), though simply functions, have multiarguments. Thus a technique must be devised whereby the arguments (or the addresses of the arguments) can be transmitted to the subprogram. The technique to be described here is essentially that employed in KINGSTRAN,[†] and is applica-

[†] J. Field et al., *Kingston FORTRAN II,* 1620 Users Group Conference, Chicago, 1964.

ble not only to functions but also to subroutine linkages. The argument list is scanned from left to right, and the addresses of all arguments are placed in a sequential list, preceded by the address of the instruction to be executed immediately after the execution of the subprogram and succeeded by a terminating mark (to be indicated herein as $\neq$).

Thus the main line program may transmit the address of the first item in the list and then branch to the subprogram, the subprogram being responsible for picking up the arguments. In a single address machine, the symbol table routine should provide two addresses: (1) the address of a word in the object time data table where the address of the first element in the list is stored, and (2) the address of the first instruction in the subprogram. Since the subprogram is relocatable, both these addresses may need to be referenced indirectly, the address of the entry point to the subprogram being entered through a transfer vector.

Since the arguments may be expressions, the values of these expressions must be placed in temporary storage locations and the addresses of these temporary locations placed in the argument transfer list. Using this method of linkage requires that all such argument expressions be evaluated before the address of the list is given to the subprogram. Thus the address of the list may be transferred through storage in the accumulator.

Let us introduce the following commands:

1. MPT Mark Place and Transfer. The execution of this instruction causes the address register of the computer to be set to the address in the operand portion of the instruction and a return address register to be set to the address of the next instruction.

2. BB Branch Back. This instruction transfers control to the instruction with an address stored in the return address register.

3. B Branch. A standard branch (or jump) instruction defines the next instruction to be executed in its operand portion.

4. DA Define Address. This is not truly an instruction but a declarative to the assembler to place the address defined in the operand portion of the symbolic instruction in the next word available in the memory.

Let us now consider the linkage to be generated when an arithmetic expression references a function. For example, consider the function call

MAX(A, B+C, E/D, F)

where it is meant that the maximum value of the four arguments is to be returned to the calling statement, and where the number of arguments in the list is unlimited. Each argument must be considered primarily as an expression and the result stored in a temporary storage location if any arithmetic operators are included. Thus in this situation, the delimiting commas and the closing parenthesis are to be considered as expression end delimiters. The above expression would be compiled to the set of instructions:

```
LDA    C
ADD    B
ST     τ1
LDA    D
RDIV   E
ST     τ2
LDA    *+2     [or JMS MAX−1] †
B      MAX
DA     *+6
DA     A
DA     τ1
DA     τ2
DA     F
DA     ≠
```

where * means "this address" and MAX[†] is a reference to the first instruction in the function MAX.

When an argument is a constant, the address of that constant should not be communicated to the subprogram as a precaution against its value being altered in the subprogram. For example, if the parameter list of a subprogram contains a parameter that is to be used as both an output and an input parameter, it would be unfortunate if, by an error in programming, the value of the constant were changed. Storing the value of the constant in a temporary storage location instead of giving a direct reference to the actual location of the constant will prevent this possibility.

When a function preference is noted in the input string, the generator must take special actions in order to regard that reference as a single value. In particular, it may not be necessary to compile the linkage instructions to the function immediately. The entry in TEO may then be the pseudo-operator F, standing for "function," together with a pointer to the left-hand parenthesis of the argument list. The function name may then be regarded as an

[†] (See page 231)

operand until it is necessary for the function to be evaluated, the entry in TEO being the key to the requirement of evaluation. The link pointing to the input string will then give access to the arguments, some of which may need evaluation themselves. This technique then conforms to the general algorithm for the evaluation of expressions containing simple variables and may save on the use of temporary storage locations by only evaluating functions as they are required. On the other hand, it is feasible that a preliminary scan of the statement to locate function references, evaluate these functions, and store the results in temporary accumulators will achieve the same effect at object time. If this course is chosen, then a temporary storage location must be available for the result of each function reference. After these evaluations, the input string must be modified to remove the function names and the list of arguments, replacing these items by the address of the temporary storage location. These special actions necessary for compiling statements within a subprogram that include references to formal parameters will be considered in the next chapter.

Problems

8.5 Amend the hierarchy list for simple arithmetic operators to include the special operators involved in subscripting computations.

8.6 Develop the production rules for the subscripting algorithm.

8.7 Extend the hierarchy tables of Problem 8.5 to include function references within arithmetic expressions and develop the production rules for compiling the instructions for the evaluation of arguments of a function reference and the linkage.

9

Compiling within a Subprogram

The subprogramming in algebraic languages serves two main purposes. In the first place, a subprogram conserves both memory space and programmer time when a particular operation or set of operations must be performed repeatedly under different circumstances and with varying sets of data. The programmer may consolidate his coding into a single set of statements and then execute that set by either using a single statement or including a reference to that subprogram in some other statement. Secondly, subprogramming enables the creation of program libraries where the programmer can find programs already coded by others. This provision may be available in one of two forms: Either the executive operating system of the computer may be programmed to fetch the subprogram from auxiliary storage where it is kept in machine language or the computing center may keep a card library of subprograms in source code available for inclusion by the programmer.

Recognizing the Problem

To use a subprogram, the programmer must follow one of two courses: He must provide, from a library or by creating it himself, a subprogram in source code which requires compilation, or he must rely on the operating system to provide a machine language subprogram from its own library. In either case, the compiler must provide the instructions, within the object code produced from the calling program, to link the calling program to the subprogram, to provide a means of return from the subprogram to the calling program, and to transmit the data to be used to the subprogram. When an algebraic language is to be developed to operate under an already existent executive system, the compiler originator may have no choice in the design of the linkage instructions to system library subprograms; for the sake of consistency, he should use the same linkage instructions to link into subprograms that were defined in source code by the programmer.

In general, references to subprograms in FORTRAN source programs may be recognized by the compiler by default. Since a subscripted variable has the same syntactical form as a reference to a function, the symbol table routine must provide to the appropriate generator the information that the reference is being made to an item that has not been defined in a DIMENSION statement. By default, this is to be considered a reference to a function subprogram. A reference to a subroutine subprogram is specific in that reference is made through a special calling statement.

In either event, the compiler will, at the recognition of an END statement, have in its possession the necessary information to determine which subprograms are required by the program being compiled. However, it cannot tell at that time which of these subprograms are to be obtained from the library and which are to be provided in source code by the programmer. Thus only when the entire job is compiled, can the system decide which subprograms the programmer has assumed are available in the system library. Also when the decision is postponed until this time, the system is provided with some subprograms whose names could be the same as those stored in the library; thus if the system removes the names of required subprograms from its list as they are provided, there will be no confusion in the naming. For some peculiar reason, this is a serious shortcoming of many systems.

When the programmer requests the use of a library subprogram, the task of the compiler is influenced not only by the form of the standard linkage, but also by the manner in which the subprogram is loaded into memory and the tasks which are fulfilled by the loader. In particular, since previously compiled subprograms are stored in machine language form, they must be programmed (or compiled) so as to be relocatable in memory. To insist that any particular subprogram be placed in memory at the same set of addresses each time it is loaded will so restrict the use of available memory that inefficient systems will result. For example, assume that a system library contains 50 subprograms, each consisting of about 200 words. If it were possible for a single program to call upon any one or all of the subprograms and if it could be shown that in any one program the use of one subprogram would not destroy the usefulness of any other subprogram, then 10,000 word positions would have to be permanently reserved for their possible storage. On the other hand, if the subprograms were to be stored in the same locations each time they were used, this would alleviate the problem of linking to them from the calling program.

The problems of creating relocatable machine codes will not be considered here since the implementation of this option is machine dependent. However, the loader of relocatable programs is required to provide data which will be resident during the execution of the object program and which may not be considered a part of the function of the loader. As a general rule, the linkage instructions in the calling program must refer to the entry point to the subprogram in order to execute that subprogram. At the time of compiling a subprogram reference, the final storage location of that subprogram may not be known. This is not, in fact, a disadvantage when it is realized that references may be made to subprograms by the use of a parameter in a subprogram, without the compiler being aware of the request. Thus at compile time, an instruction may be generated which references an address in the object time data table indirectly and into which address will be placed the actual entry address to the subprogram.

When, as part of the job, the subprogram being referenced is provided in source language, the compiler will arrange to store the object time address in this location. When the subprogram is to be provided by the system, the loader must undertake this task.

Provision of Data to the Subprogram

Now that a means of transferring control from the calling program to the subprogram has been established, it is necessary to provide a means for transferring data. In particular, since the relationships between the names of arguments in a reference to a subprogram and the formal parameters are not indicated in the defining statement, the references to parameters in the subprogram are not linked directly to the object time data table entries of the arguments. These links must be forged at object time.

On recognition of the defining statement of a subprogram, the compiler sets aside a vector in which the addresses of the arguments are to be stored by the linking instructions. All references to formal parameters within the subprogram would then be indirectly addressed to these locations. The order of the formal parameters in the defining statement is assumed to be the same as the arguments in the calling statement and thus there will be a one to one transference of addresses from the calling program to the subprogram vector. However, the calling program is not certain as to the location of the subprogram, and thus several instructions would be required in the calling program which would otherwise be superfluous. Thus if the task of transferring the addresses is assigned to the subprogram itself and the calling program provides the address of a similar list from the data table, the subprogram can access the data by transferring the addresses only when it is absolutely necessary.

In several computers, the linkage to a subprogram may be accomplished by the use of special instructions which provide some of this information. For example, in the PDP8, the JMS instruction (JUMP TO SUBPROGRAM) transfers control to the address defined in its operand plus one and places the address of the instruction to be executed on return in the word defined in its operand position. Thus if this instruction is used in a slightly different manner than that for which it was defined, the operations described above may be accomplished. At compile time, the instruction to jump to the subprogram will be generated, followed by a list of the argument addresses. At object time, the subprogram may utilize the address that is provided as the return address, as the address of the first item in this list of argument addresses and may collect them for transfer to its own vector. Further, if the return address is included in the list, it may be picked up as a standard piece of data and stored in a special location in the vector. Thus the following FORTRAN would compile as shown alongside:

```
CALL SUB(A,B,I,J)                 JMS   -SUB
                                  DA    A
                                  DA    B
                                  DA    I
                                  DA    J
                                  DA    (Return Address)
                          SUB     DA    O             (Address to be
                                  . . .               filled in later by
                                  . . .
                                  . . .               the loader)
SUBROUTINE SUB (X,Y,K,L)          DA    O             (Address to be pro-
                                                      vided by the JMS
                                                      instruction)
                          SUBENT  LDA   -SUBENT+1
                                  ST    X
                                  LDA   SUBENT-1
                                  INA   =1
                                  ST    SUBENT-1
                                  LDA   -SUBENT+1
                                  ST    Y
                                  LDA   SUBENT-1
                                  INA   =1
                                  ST    SUBENT-1
                                  . . .
                                  . . .
                                  . . .
```

When the machine does not provide this special type of linkage instruction, the instruction must be simulated by the use of a normal unconditional branch with, for example, the address of the argument list stored in the accumulator. An example of this form of linkage was shown in Chapter 8.

Storage Allocation

Using a special vector of argument addresses within the subprogram is equivalent to reserving storage for the actual values of the parameters of the subprogram. To some extent, this storage may be saved by using the same area of memory for this purpose by each subprogram, provided that the subprogram does not reference some other subprogram that utilizes the same area. For example, in the IBM 1620, the low order 100 digit positions are not generally used for the program and are supposedly reserved for console input storage. If the compiler writer can be sure that this storage will not be used for this purpose (and, in general, he has sufficient control to enforce such a requirement), then this area can be used for the storage of argument addresses. Further, in this particular machine, data can be transferred as complete records and are not restricted to the movement of single words of data; thus if the addresses are arranged at compile time to form a single record, the addresses may be transferred from their usual storage area to this particular area by a single instruction.

Though the amount of storage required for the address of a simple variable is equal to that required for storing the value of the argument, this is not the case for arrays. As far as the calling and defining statements are concerned, the only data transferred between the calling program and the subprogram are simple variables. Only when the DIMENSION statement in the subprogram is encountered, is it realized that a parameter is a reference to an array. However, since addresses and not the actual data are being communicated to the subprogram, there is no need to provide storage for an argument array in the subprogram.

However, the calling program does not possess the information that an array is to be transmitted and, in fact, this knowledge is kept from the compiler so as to allow the transmission of portions of arrays. That is, the name that appears in the argument list may not be the name of the whole array or even the first element in the array: If a portion of an array is to be transmitted, the argument is the first element in the portion to be transmitted. This creates a problem as to which address should be transmitted. As has been pointed out in previous chapters, the most efficient form of subscripting algorithm may be constructed by choosing the base address of an array as the address of the fictional element with zero subscripts. However, the definition of the base address depends on the dimensions of that array. In the case of providing an address to a subprogram, there is no rule prescribing that the dimensions of the array, which is the parameter in the subprogram, should agree with those of the argument, either in size or number of subscripts. Thus to transmit the base address of an array as defined in the calling program may not coincide with the address that would have been created in the subprogram using the dimensions stated there. Further, if the whole array is not intended, the argument may not be the first element, but the first element of the portion to be transmitted. To overcome these difficulties, the address of the first element in the array will be transmitted when only the name of the array appears in the argument list, and the actual address of the element, when a particular element is mentioned.

In the following three examples, the address list to be transmitted to the subprogram will be identical.

Case 1.

```
      DIMENSION A(4,5,6)
      . . .
      . . .
      CALL IT(. . . ,A, . . .)
      . . .
      . . .
      SUBROUTINE IT(. . . , X, . . .)
      DIMENSION X(100)
      . . .
      . . .
```

Case 2.

```
      DIMENSION A(4,5,6)
      . . .
      . . .
      CALL IT(. . . , A(1,1,1), . . .)
      . . .
      . . .
      SUBROUTINE IT(. . . , X, . . .)
      DIMENSION X(100)
      . . .
      . . .
```

Case 3.

```
      DIMENSION A(4,5,6)
      . . .
      . . .
      CALL IT(. . . , A(1,1,1), . . .)
      . . .
      . . .
      SUBROUTINE IT(. . . , Y, . . .)
      DIMENSION X(100)
C     THE NAME Y DOES NOT APPEAR IN A
C     DIMENSION STATEMENT.
      . . .
      . . .
```

The Effect of Parameters on the Generators

When data addresses are transmitted as described, the generators for statements occurring within subprograms must be cognizant of formal parameters so as to generate special forms of instructions or special instructions to take care of the fact that the actual data values are not immediately available. When the parameters of a statement are formal parameters of the subprogram, the generator need only replace the address of the variable in the generated code by an indirect address to the location at which the address of the parameter is stored. When a variable name does not occur in the parameter list, the normal process of reserving storage in the object time data table and referencing that location directly may be followed. Thus the simple assignment statement

```
X = A
```

which appears in a subprogram, and within which is a reference to the parameter A, will be compiled as

```
LDA    -A
ST      X
```

where A is the address, in the parameter address list, at which the address of the argument corresponding to the variable A has been stored.

In the case of subscripted parameters, the whole subscripting expression, rather than only a part of the algorithm, must be evaluated at compile time. That is, the total expression

$$\text{Address of element} = \text{Address } (A(1,1,\ldots 1)) - \sum_{i=1}^{N} p_i\,(s_i - 1)$$

However, the compiler writer can be assured that if a formal parameter is defined within the subprogram as an array, there is likely to be more than one reference to an element of that array within that subprogram. Thus the compiler writer must evaluate the advantage of requiring the execution of the algorithm each time a reference is made to an element of that array compared to the advantage of adding to the subprogram an initialization section, which would be prompted by the recognition of a parameter name in a DIMENSION statement and would consist of a set of instructions to compute the address of the base element in the array. That is, perform the computation

$$\text{Base Address} = \text{Address } (A(1,1,\ldots 1)) + \sum_{i=1}^{N} p_i$$

and store this address in the parameter address list in place of the address provided by the calling program. After this computation has taken place, the subscripting algorithm described previously can be incorporated into the subprogram with the minor change that the base address must be picked up at object time and cannot be provided as a constant by the compiler.

This simple method for providing the address of an element when the base address of the array is unknown at compile time may be complicated when a particular algebraic compiler allows the definition of dynamic arrays —that is, DIMENSION statements in which the sizes of arrays are defined by variables instead of the more standard constants. The FORTRAN specification states that these dimensions may only be formal parameters themselves but does not specify whether the values of these parameters may be altered during the execution of the subprogram. More important than the source of the value of the dimensioning variable (that is, whether it is a local variable or a parameter) is whether or not the size can be changed during the execution of the subprogram. If the size is inviolate once the subprogram has been entered, then the initialization process may be introduced into a subprogram with only minor changes in the generator. If the size can be altered, then the location of an element must be computed at each reference.

Although parameters of a subprogram may only be specified as variables, the arguments to be used are not so restricted. In particular, each argument may be considered an expression, the address of the result of the evaluation of that expression being provided to the subprogram. However, it would be a waste of execution time, if the values of simple variables or constants were moved to temporary storage locations before a subprogram were entered. Further, since variables that are arguments may also be used as output variables for the subprogram, the resultant values would have to be moved back to their normal locations after execution of the subprogram. To facilitate this two-way communication, it is simpler and more expedient to provide the actual address of the value of a variable to the subprogram. It is then the programmer's responsibility to ensure that any parameter that is provided for both input and output in a subprogram is provided with a means of communication in the calling program.

Thus, if an argument is defined as an expression, it may only be used as an input value since the value of an expression is not stored permanently and has no place in the object time data table. If the programmer makes an error, the output value from the subprogram is placed back into the temporary storage location where the result of the evaluation of the expression was originally placed and the remaining data are unaffected. However, if the programmer, by error, uses a constant where the subprogram expects to find an argument to be used for input and output, the value of the constant will be altered in the data table. For example, in the following program, the location in the data table that contains the integer constant 1 will contain 2 after the execution of the subprogram.

```
. . .
. . .
. . .
CALL ANY(A,1)
. . .
. . .
SUBROUTINE ANY(X,I)
. . .
. . .
I=I+1
. . .
RETURN
END
```

Since there are programs where this error would not impede the execution of the remainder of the program, it is suggested that when a constant appears in an argument list, the value be transferred to a temporary storage location so that any change in that value does not affect the rest of the program.

Often it is necessary for the argument list (and hence the parameter list) to be considered as unlimited in the number of arguments. For example, in the case of the MAX function which provides to the calling program the value of the argument in the list which is greatest, it is desirable for the number of arguments to be defined at object time. One solution to this dilemma is to EQUIVALENCE the arguments to elements in an array and then provide that array to the subprogram as a single argument.

If the function is written in machine language instead of algebraic code, the programmer can provide the instructions necessary to pick up the argument addresses. For example, if the compiler is written so that a peculiar word is added to the argument address list, then the subprogram may pick up argument addresses until this "stopper" is located. For example, if the stopper word is a word containing the address 7777, which is placed in the argument address list immediately following the return address entry, then the MAX function † may be written as the sequence of instructions:

```
        DA    0
MAX     LDA   CONST         Initialize maximum value
        ST    VAL
        LDA   -MAX+1
        ST    RET           Assume entry is return address
        LDA   MAX-1
        INA   =1            Increment address in list
        ST    MAX-1
        LDA   -MAX+1        Pick up address
        ST    STOP          Assume to be stopper
        SUB   C7777         Check for stopper
        BNZ   *+3           Jump unequal
        LDA   VAL           Load maximum value
        B     -RET          Return
        LDA   VAL
        SUB   -RET          Test argument against last
                              candidate
        BP    *+3           Jump if not larger
        LDA   -RET          Store new candidate
        ST    VAL
        LDA   STOP          Move up last address
        ST    RET
        B     MAX+4
C7777   DA    7777
CONST   DA    -77777777
VAL     DA    0
RET     DA    0
STOP    DA    0
```

Cascading Calls

Since one subprogram may reference another subprogram ad infinitum, so long as there is no reference to any one subprogram more than once, the instructions generated in a subprogram on the recognition of a call to another subprogram must be handled separately. If the particular computer being used as a target permits cascading indirect addressing, then argu-

† For an example of the calling routine to this function see p. 219.

ments within the subprogram that are also parameters may be listed as indirect addresses. Thus during the operation of this second level subprogram, any reference to a formal parameter automatically causes the cascaded indirect address to be picked up. However, in machines that do not permit cascading indirect addressing, the instructions must be inserted into the sequence of object time calling instructions so that the address in the argument list is always direct. That is, these will be inserted as a two-instruction sequence that will load the original address into the accumulator and then store this value in the argument list. Thus, if the source statement is

```
CALL SUB(A,B,I,J)
```

where B and I are parameters, then the generator must provide the instructions to chain through one level of indirect addressing to obtain the address of the original argument and place this address in the argument list. That is,

```
                LDA     —B
                ST      ARGSUB2
                LDA     —I
                ST      ARGSUB3
                JMS     —SUB
                DA      A
ARGSUB2         DA      0
ARGSUB3         DA      0
                DA      J
                DA      (return address)
                DA      7777
SUB             DA      0     (Address to be filled in later)
```

FORTRAN-type compilers do not permit the recursive calling of subprograms and thus do not provide the facilities either for a subprogram to reference itself directly or for a second level subprogram to reference back to its calling program. This feature is standard for ALGOL-type languages and thus special facilities must be provided by the compiler for this purpose. In particular, the address lists must be placed into a push down list so that as the subprogram is referenced repeatedly, the list of addresses of the arguments are saved. Further, as a pass through the subprogram is completed, the list must pop up the last set of argument addresses. In addition, no instructions or permanent addresses in the subprogram may be altered during the execution of the subprogram, and any intermediate results must be placed in a push down list. However, the organization of this system requires that the push down list storage areas be reserved in advance, irrespective of the actual use and extent of recursion.

Function Names as Arguments

When a function name is used as an argument in the calling statement of a subprogram, the entry address of the function is to be regarded as the value of the argument and, provided that this address is stored in the object time data table, the organization of the argument address list is unaffected. However, the calling program must have the ability to recognize this form of argument, particularly since the usual determination technique is unworkable in this instance. That is, in general, a function reference is recognized syntactically as being one of two possible accretions, either a subscripted variable or a function reference. However, the information stored in the symbol table enables the compiler to recognize the subscripted variable since that name will have been previously defined in a DIMENSION statement. By default, an accretion that takes the form of a subscripted variable but has not been defined in a DIMENSION statement is assumed to be a function reference. However, when a function name is used as an argument, no argument list is appended, and, in fact, if one were it would be assumed that the value of the function was to be used as the argument value. Thus, with no previous definition, a function name used as an argument is syntactically identical to a variable name. Consequently, FORTRAN requires the definition of function names that are to be used in this manner. This is achieved by means of the EXTERNAL statement.

It is worth pointing out that many programmers are confused by the keyword EXTERNAL. By connotation, it would seem that this statement defines those functions that are external to the program, that is, those functions that are to be referenced by that program. This is the opposite to what is intended. Perhaps the solution to this pragmatic error would be to invent the keywords TYPE FUNCTION.

In the subprogram, a reference to a function is recognized in the normal fashion except when it is used as an argument; then the language requires the presence of an EXTERNAL statement within the subprogram. In an actual subprogram call, the address of the function referenced may be picked up from the argument address list instead of providing a special data table entry for the storage of the address of the entry point to the subprogram. Thus the following statement, which references a subprogram that is named as a parameter, would compile:

```
CALL DUMMY(I,J)    JMS   -DUMMY
                   DA    I
                   DA    J
                   DA    (Return Address)
                   DA    7777
```

where the mnemonic address DUMMY is that address in which the address of the parameter function is stored.

When a function reference that is also a parameter is used in an argument list, the appropriate indirect chaining must be accomplished by object time instructions in the same manner as with transferring parameters that are variables.

Temporary Storage

Temporary storage locations are needed in subprograms in the same manner as in main-line programs, but the choice of locations is complicated by the fact that a function subprogram may be called from within an arithmetic statement that is also utilizing storage temporarily. It would seem possible, at first glance, to utilize those storage locations that are not being used in that calling statement, but this would involve the use of different storage locations by the subprogram each time it was used. Since this is inconvenient and time consuming, a separate set of storage locations that do not overlap with those used by the calling program or any other subprogram must be provided for each function subprogram.

Subroutine subprograms are not restricted in their use of temporary storage locations since a subroutine subprogram is called by a unique statement. In the calling statement, the only use of temporary storage locations when the subprogram is entered will be to store the results of arguments that are expressions. Even though these expressions may require temporary storage for their evaluation, the compiler can arrange to have the storage locations used for this purpose chosen from the set remaining after locations have been reserved for possible storage of the evaluated arguments. Thus a subroutine subprogram may utilize the remaining set of temporary storage locations after this reservation for evaluated arguments, this number being determined at compile time from the subprogram defining statement.

Temporary storage is also required in a function subprogram for use with the function name being used as a variable. That is, by definition of a function subprogram, there must be one statement within the subprogram that assigns a value to a location which is given the same name as the function. This means that in a function subprogram, the compiler must be capable of recognizing the name of the function in this context and reserving storage for this variable. Further, since the result of evaluating a function is not stored in a location in the object time data table and thus is not directly available to the programmer, but rather is to be made available to the calling statement as a value for further manipulation, this value must be placed into the accumulator when a RETURN statement is executed. Thus the generator for a RETURN statement does not simply produce a branch indirectly to the address of the next instruction to be executed, but must add, in the case of a function subprogram, a load instruction to extract the result from the location chosen for the storage of the values of the variable that carries the same name as the function. For this purpose, the generator for the defining statement of a function subprogram must reserve storage under the name of the function, but the name is to be utilized as a simple variable of the same mode as the function. However, this generator must not define the variable as carrying a value under the normal rules of defining a variable.

Summary

The special productions necessary for compilation within a subprogram result from the techniques of linking the calling program with the subprogram and the methods of providing the arguments to the subprogram. The linkages between the different types of subprogram should be consistent so as to minimize variances within the compiler. For example, since it is known (by definition of the language) that function subprogram arguments are all input values, it would be possible to provide only the actual values of those arguments as input to the subprogram. However, with the subroutine subprogram, the compiler cannot determine, when the calling statement is being compiled, which arguments are to be used as input parameters and which are to be used for storing results. Thus the linkage mechanism for a subroutine subprogram requires the more general technique which for ease of compilation is used for all subprograms independent of their type or origin. Although in any one situation, a compile writer can show that a peculiar form of linkage is more efficient, the genera technique will ease the task of the compiler and thus the programmer

There are many ways of implementing a compiler, several of which wil depend for their utility, on the external environment of the system, and or the fancies of the originator.

APPENDIX A

The Logical Flow of the IBM 1401 Processor[†]

Phase 01—System Monitor

Note: This phase is resident in the computer memory throughout the compilation. When any other phase has completed its task, control is transferred back to the monitor.

1. Brings in next phase from the current input unit.
2. Ensures that between phases, no extraneous material is left which would jeopardize the execution of the incoming phase.

Phase 02—Loader

1. Stores the information on the control card that precedes each source deck.
2. Checks the storage information on the control card against the available memory unless the program is not to be compiled for execution on this machine.
3. Reads in the source program and stores it in memory, appending to each statement a three-character position for a sequence number and a one-character position for a statement type code.
4. Deblanks the statements except in the Hollerith fields of FORMAT statements.
5. Collects continued statements and checks for an overabundance of the same.
6. Checks for special input statement characters and converts any, if found, to special internal codes.
7. Places special delimiters around each statement. (Because of deblanking, the statements are not of equal length.)
8. Generates a STOP statement after the last statement.

[†] L. H. Haines, "Serial Compilation and the 1401 FORTRAN Compiler," *IBM Systems Journal,* Vol. 4, No. 1, 1965, Form No. 321-0002. Reprinted and edited by permission from *IBM Systems Journal* © 1965, by International Business Machines Corporation.

Phase 03—Scanner

1. Determines the statement type of each statement and inserts the appropriate code in the appendage to the statement.
2. Numbers each statement in sequence.

Phase 04—Sort I

Tests available memory to determine whether each statement can be expanded by three characters. If this is not possible, the compilation is terminated after a message is output, indicating that the object program is too large.

Phase 05—Sort II

By expanding each statement by three characters, statements of the same type are linked so that each statement has the address of the next statement of the same type appended to it.

Phase 06—Sort III

The source program is sorted by statement type and shifted to low storage.

Phase 07—Insert Group Mark

The delimiter that separates the statement from its appendage is replaced by a group mark.

Phase 08—Squoze

1. Keywords are eliminated from the source statements, and the statements are squeezed to expand the available storage.
2. Statements with invalid keywords are eliminated from the program and appropriate error messages printed.

Phase 09—Dimension I

1. A table containing the names of variables mentioned in the chain of DIMENSION statements is constructed in high memory.
2. Each table element contains: (a) the array name, (b) the number of dimensions, (c) the size of each dimension, and (d) a space for control characters and data to be generated in the EQUIVALENCE phases and the second DIMENSION phase.

Phase 10—Equivalence I

1. Checks all arrays mentioned in EQUIVALENCE statements to ensure that they have occurred in DIMENSION declarations.

2. Adds simple variables that occur in EQUIVALENCE statements to the table of arrays generated in phase 09. These variables are inserted into the table as if they were single element arrays.

Phase 11—Equivalence II

Computes the offsets of equivalenced arrays and notes the relationships between arrays (that is, implicit equivalencing as a result of the mentioning of a single array in more than one EQUIVALENCE group).

Phase 12—Dimension Phase II

Arrays are assigned object time addresses.

Phase 13—Variables I

1. The source statements are scanned for variable names.
2. Simple variables are tagged for processing in phase 16.
3. Subscripted variables with constant subscripts are replaced by object time addresses.
4. Subscripted variables with variable subscripts are replaced by the computation required at object time to compute the location of that element.
5. Array names appearing in lists are replaced by two memory addresses denoting the limits of the array when no subscripts are appended to the name.
6. Array names appearing without subscripts in other places are replaced by the address of the first element of the array.

Phase 14—Variables II

The entire source program is moved to high memory, leaving room for subsequent phases. The remaining storage is then cleared for tables including that generated in phase 12.

Phase 15—Variables III

This is not a self-sufficient phase, but is resident during phases 16 and 17. Phase 15 is a housekeeping routine.

Phase 16—Variables IV

1. The compiler scans input statements and the left-hand side of assignment statements for simple variables. Each unique variable is assigned an object time address.
2. All variables in statements are checked against the object time address table, and when a match is found, the object time address is substituted for the variable name. When a match is not found, it is assumed that the variable is undefined.

Phase 17—Variables V

A check is made for unused variables.

Phase 18—Constants I

Constants in the source program are extracted and converted to internal mode with truncation if necessary.

Phase 19—Constants II

This phase is the same as phase 14. The table of simple variables is destroyed.

Phase 20—Constants III

The constants are assigned object time addresses at the low end of memory. The constants are then placed in these locations and are replaced in the source program by the addresses.

Phase 21—Subscripts

Subscripts that require object time computation (that is, consist of expressions) are reduced to a set of parameters.

Phase 22—Statement Numbers I

All statement numbers appearing in the source program are converted to a unique three-character code.

Phase 23—Format I †

All input/output statements are checked against the FORMAT statements to ensure that all the FORMAT statements are necessary. Unreferenced FORMAT statements are discarded.

Phase 24—Format II †

The object time FORMAT strings are developed and stored in the low end of memory.

Phase 25—Lists I

Lists are compared to eliminate duplicates and thus to optimize object time storage.

† The authors of this system named phases 23 and 24, Tamrof I and II.

Phase 26—Lists II

The object time lists of addresses and instructions (to compute array element locations) are developed and stored at the low end of memory.

Phase 27—Lists III

Each input/output statement is reduced to the address of the first item in the list string (if present), the address of the FORMAT string, and the logical unit number.

Phase 28—Statement Numbers II

This phase is the same as phase 14.

Phase 29—Statement Numbers III

The three character codes of statement numbers appearing within statements are stored in a table.

Phase 30—Statement Numbers IV

The statement numbers appearing as identifiers in statements are checked against the table of statement numbers generated in phase 29. When a match is found, the sequence number generated in phase 03 is placed in the table. Undefined and multiply defined statement numbers are also checked.

Phase 31—Statement Numbers V

Unreferenced statement numbers are noted.

Phase 32—Input/Output I

The linkage to the object time FORMAT routine is inserted in each input/output statement prior to the data generated in phase 27.

Phase 33—Arithmetic I

This phase is a resident housekeeping phase for phases 34–38:

1. The unary minus and exponential operators are converted to unique one-character symbols. The unary plus is discarded.
2. Error checking takes place.

Phase 34—Arithmetic II

1. All arithmetic and arithmetic IF statements are coded by a forcing table technique (see page 192).
2. Error checking continues.

Phase 35—Arithmetic III

Initialization for phase 36 occurs.

Phase 36—Arithmetic IV

The data generated in phase 35 are scanned to optimize the number of temporary accumulators needed for each statement.

Phase 37—Arithmetic V

1. IF statement tests and exits are coded.
2. Involution routine linkages are coded.

Phase 38—Arithmetic VI

The optimization of temporary accumulators started in phase 36 is completed, and object time storage locations are assigned.

Phase 39—Input/Output II

Instructions for executing tape-manipulation commands are created, that is, ENDFILE, REWIND and BACKSPACE.

Phase 40—Computed GO TO

Computed GO TO statements with two to ten exits are coded by the use of in-line instructions. For statements with more than ten alternate exits, linkage to a system subroutine is generated.

Phase 41—GO TO

Unconditional GO TO statements in the source program are replaced by in-line branch statements.

Phase 42—STOP/PAUSE

The object time instructions to halt (for STOP) or halt and continue (for PAUSE) are generated together with the instructions necessary to display the indication number.

Phase 43—SENSE LIGHT

In-line instructions are generated to execute the sense light operations.

Phase 44—Hardware IF

The instructions to test and branch on IF(SENSE SWITCH i) or IF(SENSE LIGHT i) are generated in line.

Phase 45—CONTINUE

This phase merely collects data for later phases. It does not generate any instructions.

Phase 46—DO

1. The DO statements are replaced by: (a) an unconditional branch and (b) a set of parameters describing the elements of the DO statement.
2. An unconditional branch is prepared which will be inserted after the last statement within the range of the DO.

Phase 47—Resort I

Initializations for phase 48 are executed.

Phase 48—Resort II

A special table (the resort table) is filled with the current location of each statement in memory.

Phase 49—Resort III

The source statements are sorted back into their original order, that is, the order before the execution of phase 06. The statement number table is updated to show the current address of each statement.

Phase 50—Resort IV

The statements are relocated to occupy the places in memory that they occupy at execution time. The statement number table is adjusted to show these addresses.

Phase 51—Replace I

1. Object time instructions that contain references to statement numbers (which presently appear as code characters rather than as actual addresses) are corrected to reflect the object time addresses of the referenced statements.
2. Subscript strings are cleaned up.

Phase 52—Function/Subroutine Loader

1. The relocatable functions and subroutines called in the source program are loaded into memory.
2. A table of the starting addresses of these routines is prepared.

Phase 53—Relocatables

This is not truly a phase of the compiler since it takes no part in the translation of the source program. This phase (maybe better called a package) consists of the routines that are loaded by phase 52.

Phase 54—FORMAT routine loader

This routine loads the object time FORMAT routine.

Phase 55—Replace II

The instructions that were generated in phase 34 and that reference system routines are corrected to reflect the object time location of these routines after being loaded by phase 52.

Phase 56—Snapshot

If requested on the control card (see phase 02) a snapshot of the generated object program is printed provided that no source program errors have been recognized that would create a "NO GO" situation.

Phase 57—Condensed Deck I

If requested, and provided an "O.K." program has been produced, a condensed object program will be punched. This phase punches only the clear storage and bootstrap cards.

Phase 58—Condensed Deck II

This phase duplicates phase 59 into the object deck.

Phase 59—Fixed Routines

This package, which is duplicated into the object deck by phase 58, consists of: (a) the arithmetic routines, (b) initialization routines that set up the index registers and sense lights, and (c) the snapshot routine for use in debugging at object time.

Phase 60—Condensed Deck III

The generated object time instructions and data are punched together with the FORMAT routine, which was loaded at phase 54. Only the actual used storage is punched.

Phase 61—Geaux I

This phase prints the end-of-compilation messages.

Phase 62—Geaux II

The arithmetic routine is read into storage, and communication between this package and the relocatable routines is established. Note that the arithmetic routines were not loaded at phase 58 since there was insufficient space to contain both phase 01 (the monitor) and phase 58 while the arithmetic routine was loaded. This phase destroys the monitor.

Phase 63—Arithmetic Package

This package consists of the fixed arithmetic routines which are to be loaded by phase 62. In fact, this package is a duplicate of phase 59.

After the monitor is destroyed in phase 62, the control is transferred to the existing program which is now ready for execution. However, even though this is a load and go compiler, an object deck has been produced for subsequent executions.

APPENDIX B

FORMAC—The Formula Manipulation Compiler

This appendix takes the form of a FORMAC program in which the comments (that is, the lines that contain a C in the first column as in FORTRAN) describe the action or meaning of the statements. Since FORMAC is a parasite on FORTRAN, any normal FORTRAN statements will have no explanation. The second part of the appendix shows the output from this program.

```
C  A PROGRAM TO DEMONSTRATE THE CAPABILITIES OF FORMAC
C
      SYMARG
C  SYMARG FLAGS THE START OF THE FORMAC PROGRAM IN THE SAME
C  MANNER AS PROGRAM IN A FORTRAN PROGRAM
      ATOMIC U, V, W, X, Y, Z, P, T, AORB, ALPHA, BETA, THORC
C  AN ATOMIC STATEMENT DECLARES THAT THE VARIABLES THAT APPEAR
C  IN ITS LIST REPRESENT THEMSELVES AND NOT VALUES AS IN FORTRAN.
C  THAT IS, THESE ARE THE VARIABLES OF FORMAC THAT WILL APPEAR
C  IN EXPRESSIONS OR FORMULAS THAT ARE TO BE MANIPULATED.
      DEPEND (U, V, W, Y, Z/X)
C  SINCE FORMAC HAS THE ABILITY TO DIFFERENTIATE IT IS NECESSARY
C  TO DESCRIBE THE DEPENDENCY OF THE ATOMIC VARIABLES. IN THE
C  ABOVE STATEMENT U, V, W, Y AND Z ARE SAID TO DEPEND ON X.
      DIMENSION CARD(12), LINE(12)
      LET POLY1=P**5 + 15*P**4 + 105*P**3 + 420*P**2 + 945*P + 945
C  THE LET STATEMENT CAUSES THE VARIABLE NAME ON THE LEFT OF THE
C  EQUAL SIGN TO BE ASSIGNED TO THE EXPRESSION ON THE RIGHT.
C  AS OPPOSED TO A FORTRAN ASSIGNMENT STATEMENT, A LET STATEMENT
C  MERELY ASSIGNS A NAME TO AN EXPRESSION AND DOES NOT INFER
C  ANY COMPUTATIONS.
      LET POLY2=P**7 + 28*P**6 + 378*P**5 + 3150*P**4 + 17325*P**3
     1 + 62370*P**2 + 135135*P + 135135
C  MULTIPLY THESE TWO EXPRESSIONS
      LET POLY = EXPAND POLY1 * POLY2
C  THE EXPAND KEYWORD WILL DEPARENTHESIZE THE RESULTING EXPRESSION
```

```
C  AND COLLECT THE COEFFICIENTS OF THE POWERS OF THE VARIABLE P
       LET POLY = ORDER POLY, DEC, FUL
C  THE KEYWORD ORDER WILL CAUSE THE EXPRESSION TO BE SEQUENCED.
C  BY VIRTUE OF THE WORD DEC, THE SEQUENCING WILL BE IN DECREASING
C  POWERS OF THE VARIABLE.
C  THE NEXT FOUR STATEMENTS ARE A STANDARD SET TO OUTPUT AN
C  EXPRESSION WHICH HAS BEEN STORED UNDER THE NAME OF A VARIABLE.
       BEGIN = 0.0
    10 LET BEGIN = BCDCON POLY, LINE, 12
C  BCDCON IMPLIES CONVERSION FROM FORMAC INTERNAL REPRESENTATION
C  TO BCD FOR OUTPUT. THE RESULT OF THE CONVERSION IS STORED
C  BOTH IN BEGIN AND LINE EXCEPT THAT LINE IS FILLED WITH DIFFERING
C  PORTIONS OF POLY DURING EACH PASS THROUGH THE LOOP.
       WRITE(6,920)(LINE(J), J=2,12)
       IF(BEGIN.NE.0.0)GO TO 10
C  DIFFERENTIATE THE RESULTANT PRODUCT
       LET DERIV = FMCDIF(POLY, P, 1)
C  FMCDIF IS THE FORMAC DIFFERENTIATION ROUTINE
C  THE PARENTHESIS CONTAINS THE NAME OF THE EXPRESSION TO BE
C  DIFFERENTIATED, THE VARIABLE OF DIFFERENTIATION AND THE ORDER
C  TO WHICH DIFFERENTIATION IS TO BE CONTINUED. THAT IS, IN THE
C  ABOVE STATEMENT, POLY IS TO BE DIFFERENTIATED ONCE WITH
C  RESPECT TO THE ATOMIC VARIABLE P
       LET DERIV = ORDER DERIV, DEC, FUL
    20 LET BEGIN = BCDCON DERIV, LINE, 12
       WRITE(6,920)(LINE(J), J=2,12)
       IF(BEGIN.NE.0.0)GO to 20
C  EVALUATE THE RESULTANT POLYNOMIAL FOR P = -3.6
       LET VAL = EVAL POLY, (P, -3.6)
C  THE EVAL VERB DEFINES THE EVALUATION OF THE EXPRESSION
C  USING THE NUMERICAL VALUES OF THE ATOMIC VARIABLES
C  LISTED IN THE PARENTHESES.
C  THE RESULT OF THIS OPERATION PLACES A NORMAL FORTRAN NUMBER
C  IN THE VARIABLE VAL. THUS THE RESULT MAY BE OUTPUT WITHOUT
C  ANY SPECIAL INSTRUCTIONS.
       WRITE(6,950) VAL
C  EVALUATE THE POLYNOMIAL FOR P = -3.7
       LET VAL = EVAL POLY, (P, -3.7)
       WRITE(6,950) VAL
C  FIND THE 4TH DERIVATIVE OF THE INTEGRAND IN THE INTEGRAL
C  FORMULATION OF THE BESSEL FUNCTION J1(X)
       LET FOURTH = EXPAND FMCDIF(FMCCOS(T - X*FMCSIN(T)), T, 4)
C  TO PREVENT ANY CONFUSION BETWEEN THE SINE FUNCTION OF FORTRAN
```

```
C  AND THAT OF FORMAC, THE LATTER IS GIVEN THE DISTINCT NAME
C  OF FMCSIN. SIMILARLY FOR ALL OTHER FUNCTIONS.
C  PRINT THE DERIVATIVE FUNCTION.
         LET FOURTH = ORDER FOURTH, INC, FUL
C  THIS EXPRESSION IS TO BE ORDERED IN INCREASING POWERS
         BEGIN = 0.0
      45 LET BEGIN = BCDCON FOURTH, LINE, 12
         WRITE(6,920)(LINE(J), J=2,12)
         IF(BEGIN.NE.0.0) GO TO 45
C  READ EXPRESSIONS FROM CARDS AND DIFFERENTIATE THEM
      41 READ(5,900)CARD
         BEGIN = 0.0
         LET EXPR = ALGCON CARD, BEGIN
C  ALGCON CONVERTS FROM BCD TO FORMAC INTERNAL MODE
         LET ANS = EXPAND FMCDIF(EXPR, X, 1)
         LET ANS = ORDER ANS, INC, FUL
         WRITE(6,910)
         BEGIN = 0.0
      50 LET BEGIN = BCDCON EXPR, LINE, 12
         WRITE(6,920)(LINE(J), J=2,12)
         IF(BEGIN.NE.0.0) GO TO 50
         WRITE(6,930)
      60 LET BEGIN = BCDCON ANS, LINE, 12
         IF(BEGIN.NE.0.0) GO TO 60
         ERASE ANS
C  THE ERASE VERB ELIMINATES UNWANTED EXPRESSIONS FROM STORAGE
C  AND THEREFORE CONSERVES MEMORY.
         GO TO 41
     900 FORMAT(12A6)
     910 FORMAT(//1H ,25HTHE EXPRESSION READ IS . . . //)
     920 FORMAT(5X, 12A6)
     930 FORMAT(//1H ,53HTHE DIFFERENTIATION PERFORMED BY FORMAC
        RESULTS 1IN . . . //)
         END
```

The remainder of this appendix contains the results of executing the above FORMAC program. No input data were needed for the first portion of the program and that for the second part is reflected in the output.

Results

```
2.5540515E8*P+P**12.0+43.0*P**11.0+903.0*P**10.0+12180.0*P**9.0+
116970.0*P**8.0+839160.0*P**7.0+4596480.0*P**6.0+19321470.0*P**5.0
+61760475.0*P**4.0+1.4625765E8*P**3.0+2.43398927E8*P**2.0+
127702575.0$ (Note: the $ sign signifies the end of an expression)
4.86797843E8*P+12.0*P**11.0+473.0*P**10.0+9030.0*P**9.0+109620.0*P
**8.0+935760.0*P**7.0+5874120.0*P**6.0+27578880.0*P**5.0+
96607350.0*P**4.0+2.470419E8*P**3.0+4.38772943E8*P**2.0+
2.554051E8$
  4.9110000E 03
 -5.0090000E 03
7.0*X*FMCSIN(T)*FMCSIN(T-X*FMCSIN(T))-8.0*X*FMCCOS(T)*FMCCOS
(T-X*FMCSIN(T))-12.0*X**2.0*FMCSIN(T)*FMCSIN(T-X*FMCSIN(T))*
FMCCOS(T)-3.0*X**2.0*FMCSIN(T)**2.0*FMCCOS(T-X*FMCSIN(T))+10.0
*X**2.0*FMCCOS(T)**2.0*FMCCOS(T-X*FMCSIN(T))+6.0*X**3.0*FMCSIN
(T)*FMCSIN(T-X*FMCSIN(T))*FMCCOS(T)**2.0-4.0*X**3.0*FMCCOS(T)
**3.0*FMCCOS(T-X*FMCSIN(T))+X**4.0*FMCCOS(T)**4.0*FMCCOS(T-X*
FMCSIN(T))+FMCCOS(T-X*FMCSIN(T))$
THE EXPRESSION READ IS . . .
    X*2.0*21.0+X**3.0+FMCSIN(X**2.0)+FMCCOS(X)$
THE DIFFERENTIATION PERFORMED BY FORMAC RESULTS IN . . .
    2.0*X*FMCCOS(X**2.0)+42.0*X+3.0*X**2.0-FMCSIN(X)$
THE EXPRESSION READ IS . . .
    U*V$
THE DIFFERENTIATION PERFORMED BY FORMAC RESULTS IN . . .
    U*FMCDIF(V,(X,1))+V*FMCDIF(U,(X,1))$
THE EXPRESSION READ IS . . .
    AORB*FMCSIN(BETA*X+THORC)*FMCEXP(-ALPHA*X)$
THE DIFFERENTIATION PERFORMED BY FORMAC RESULTS IN . . .
    -ALPHA*AORB*FMCSIN(BETA*X+THORC)*FMCEXP(-ALPHA*X)+AORB*
    BETA*FMCCOS(BETA*X+THORC)*FMCEXP(-ALPHA*X)$
```

The above program and results were provided to the author by D. D. McCracken, for which the author is extremely grateful. However, the reader should verify the results.

Further details of FORMAC may be obtained from the following publications:

"FORMAC" (Operating and User's Preliminary Reference Manual), IBM Program Information Dept., Hawthorne, N. Y., Form No. 7090 R21BM 0016, Aug. 1965.

Sammet, J. E., and E. R. Bond, "Introduction to FORMAC." *IEEE Trans. on Elect. Comp.*, Vol. EC-13, No. 4, Aug. 1964.

APPENDIX C

A Formal Definition of Dartmouth Basic†

<alphabetic character> := A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<digit> := 0|1|2|3|4|5|6|7|8|9

<special character> := +|−|*|/|b|=|(|)|>|<|.|,|; |↑

<integer> := {*<digit>*}$_1^9$

<fraction> := .***<integer>***

<decimal number> := {*<digit>*}$_1^{n \leqq 9}$. {*<digit>*}$_0^{9-n}$

Note: A decimal number could not be defined as

<decimal number> := *<integer>*.*<integer>*

since (a) no more than nine digits are permitted in a number, whereas the above construct would allow a maximum of 18 and (b) since an *<integer>* must contain at least one digit, the form {*<digit>*}. **cannot be generated.**

<sign> := *<null>*|−|+

<exponent> := E*<sign>*{*<digit>*}$_1^2$

<number> **:={*<integer>*|*<fraction>*|*<decimal number>*}**$_1^1$
{*<exponent>*}$_0^1$

<signed number> := *<sign><number>*

<simple variable> := *<alphabetic character>*{*<digit>*}$_0^1$

<subscripted variable> :=
<alphabetic character>(*<expression>*{,*<expression>*}$_0^1$)

<variable> := *<simple variable>*|*<subscripted variable>*

<function name> := SIN|COS|TAN|ATN|EXP|ABS|LOG|SQR|INT|RND|
FN*<alphabetic character>*

<function term> := *<function name>*(*<expression>*)

<term> := *<number>*|*<variable>*|*<function term>*|(*<expression>*)

<involution factor> := *<term>* **|*<involution factor>*↑*<term>***

<multiply factor> := *<involution factor>*|
<multiply factor>{*|/}$_1^1$*<involution factor>*

<expression> := *<multiply factor>*|*<sign><expression>*|
<expression>{+|−}$_1^1$*<multiply factor>*

† "BASIC" (A Manual for BASIC, the elementary algebraic language designed for use with the Dartmouth Time-Sharing System), Dartmouth College, Jan. 1, 1965.

<*assignment statement*> := LET<*variable*>=<*expression*>

<*read list*> := <*variable*>{,<*variable*>}$_{0}^{\infty}$

<*READ statement*> := READ<*read list*>

<*number list*> := <*signed number*>{,<*signed number*>}$_{0}^{\infty}$

<*DATA statement*> := DATA<*number list*>

<*message*> := "{<*alphabetic character*>|<*digit*>|<*special character*>}$_{1}^{\infty}$"

<*print item*> := <*expression*>|<*message*>|<*message*><*expression*>

<*print list*> := <*null*>|<*print item*>{,<*print item*>}$_{0}^{\infty}${,}$_{0}^{1}$

<*PRINT statement*> := PRINT<*print list*>

<*line no 1*> := {<*digit*>}$_{1}^{3}$

<*GO TO statement*> := GO{b}$_{0}^{1}$TO<*line no 1*>

<*comment*> := REM{<*alphabet character*>|<*digit*>|<*special character*>}$_{0}^{\infty}$

<*relation op*> := >=|>|<>|<|<=|=

<*IF statement*> := IF<*expression*><*relation op*><*expression*>THEN
<*line no 1*>

<*FOR statement*> := FOR<*simple variable*>=<*expression*>TO<*expression*>
{STEP<*expression*>}$_{0}^{1}$

<*NEXT statement*> := NEXT<*simple variable*>

<*END statement*> := END

<*size*> := <*integer*>{,<*integer*>}$_{0}^{1}$

<*dimension variable*> := <*alphabetic character*>(<*size*>)

<*DIMension statement*> := DIM<*dimension variable*>
{,<*dimension variable*>}$_{0}^{\infty}$

<*DEFine statement*> := DEFbFN<*alphabetic character*>(<*simple variable*>)
=<*expression*>

<*GOSUB statement*> := GOSUB<*line no 1*>

<*RETURN statement*> := RETURN

<*statement body*> := <*assignment statement*>|<*READ statement*>|
<*DATA statement*>|<*PRINT statement*>|
<*GO TO statement*>|<*IF statement*>|
<*FOR statement*>|<*NEXT statement*>|
<*DIMension statement*>|<*DEFine statement*>|
<*GOSUB statement*>|<*RETURN statement*>|
<*comment*>

<*line number*> := {<*digit*>}$_{1}^{3}$b

<*BASIC statement*> := <*line number*><*statement body*>

<*BASIC program*> := {<*BASIC statement*>}$_{1}^{\infty}$
<*line number*><*END statement*>

APPENDIX D

Solutions to Selected Problems

Chapter 2

Problem 2.1

Define:

$$\begin{aligned}
&<\textit{odd digit}> := 1|3|5|7|9\\
&<\textit{even digit}> := 0|2|4|6|8\\
&<\textit{digit}> := <\textit{odd digit}>|<\textit{even digit}>
\end{aligned}$$

Then we may define:

$$\begin{aligned}
&<\textit{even integer}> := \{<\textit{digit}>\}_0^\infty<\textit{even digit}>\\
&<\textit{odd integer}> := \{<\textit{digit}>\}_0^\infty<\textit{odd digit}>
\end{aligned}$$

Problem 2.3

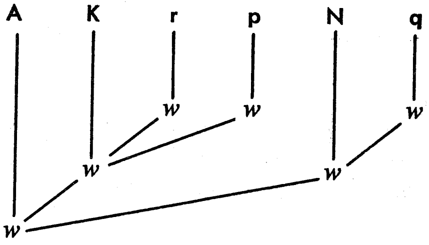
$$<\textit{sub exp}> := \{<c>*\}_0^1<v>\{\{+|-\}_1^1<c>\}_0^1|\ <c>$$

Problem 2.5

$$\begin{aligned}
&<\textit{sterling constant}> := <\textit{pounds field}>/<\textit{shillings field}>/<\textit{pence field}>L\\
&<\textit{pounds field}> := \{<\textit{digit}>\}_1^\infty\\
&<\textit{shillings field}> := 1<\textit{digit}>|1|2|3|4|5|6|7|8|9|—\\
&<\textit{pence field}> := 1\{0|1\}_0^1 2|3|4|5|6|7|8|9|—
\end{aligned}$$

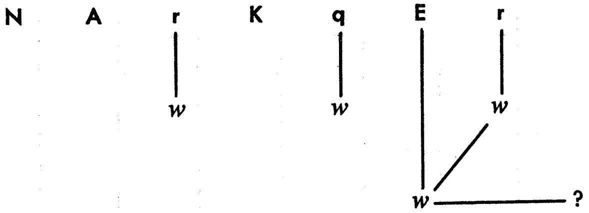
Chapter 3

Problem 3.1(a)



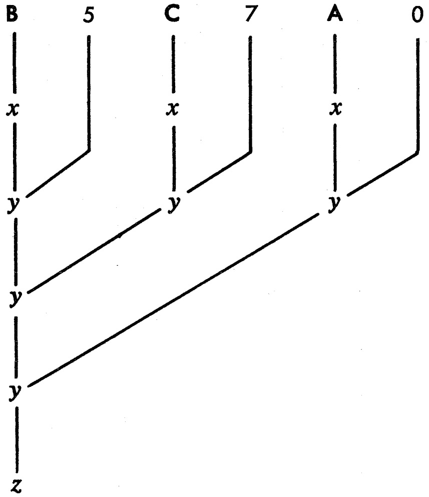
The string is valid.

Problem 3.1(e)



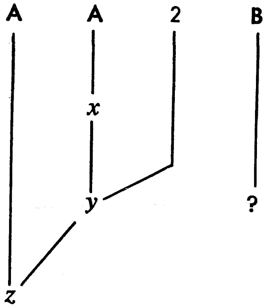
The string is invalid.

Problem 3.2(c)



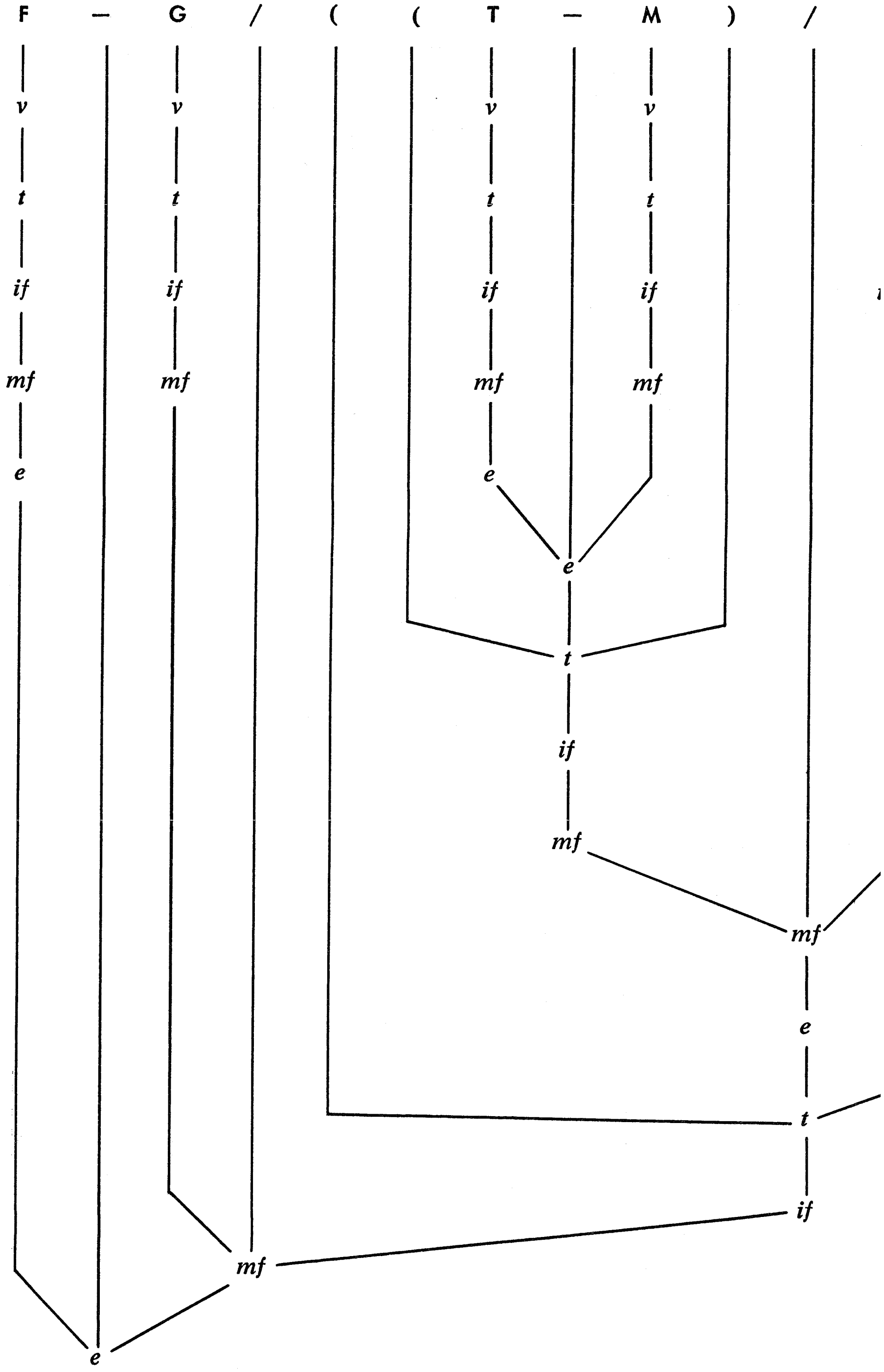
The string is valid.

Problem 3.2(d)



The string is invalid.

Problem 3.4(c)



Chapter 4

Problem 4.1(a)

$$110110_2 = 54_{10}$$

Problem 4.1(c)

$$011011_2 = 27_{10}$$

Problem 4.2(a)

93_{10} is input into two words as 001001 and 000011. Multiplying out the polynomial:

$$\begin{array}{lr}
\text{Word 1:} & 001001 \\
 & 1010\times \\
\hline
 & 001011010 \\
\text{Word 2:} & 000011+ \\
\hline
\text{Ans:} & 001011101 \\
\hline
\end{array}$$

Problem 4.2(c)

256_{10} is input into three words as 000010, 000101 and 000110.

$$\begin{array}{lr}
\text{Word 1:} & 000010 \\
 & 1010\times \\
\hline
 & 000010100 \\
\text{Word 2:} & 000101+ \\
\hline
 & 000011001 \\
 & 1010\times \\
\hline
 & 000011111010 \\
\text{Word 3:} & 000110+ \\
\hline
\text{Ans:} & 000100000000 \\
\hline
\end{array}$$

Problem 4.3(a)

Convert mantissa to an integer to ease the conversion:

$$800.46 \times 10^{20} = 80046. \times 10^{18}$$

Now $$10^{18} = 2^{18 \log_2 10} = 2^{59.8}$$

Then $$80046. \times 10^{18} = 80046. \times 2^{0.8} \times 2^{59}$$

From which we may calculate (to four figure accuracy)

$$\mathbf{80046. \times 2^{0.8} = 139280.}$$

Converting the mantissa to binary:

$$139280_{10} = 1000100000000100001_2$$

Normalizing this mantissa and truncating to five significant digits:

$$0.10001 \times 2^{18}$$

The original exponent of 2 must now be increased by the amount of the shift (which is 18), hence:

$$800.46 \times 10^{20} = 0.10001_2 \times 2^{77}$$

Finally, converting the exponent to binary mode:

$$800.46 \times 10^{20} = 0.10001_2 \times 2^{1001101_2}$$

Problem 4.4

```
      PROGRAM CATCH
      DIMENSION NAME(20),IN(80)
   23 READ(60,100)IN
  100 FORMAT(80R1)
      IF(EOF,60)20,21
   20 STOP
   21 I=1$J=1
      WRITE(61,104)IN
  104 FORMAT(//1X,80R1)
C     FIND FIRST ALPHABETIC CHARACTER
    5 IF(IN(J).LT.21B.OR.IN(J).GT.71B.OR.IN(J).EQ.60B.OR.IN(J).EQ.54B.OR
     1.IN(J).EQ.33B.OR.IN(J).EQ.34B.OR.IN(J).EQ.40B)GO TO 2
      CALL EXTRACT(J,IN,NAME(I))
      I=I+1
    2 J=J+1
      IF(J.LE.80)GO TO 5
      I=I-1
      WRITE(61,102)(NAME(K),K=1,I)
  102 FORMAT(1X,A8)
      GO TO 23
      END
      SUBROUTINE EXTRACT(J,IN,NAME)
      DIMENSION IN(80),ISAVE(9)
      DO 6 K=2,9
    6 ISAVE(K)=1H
      ISAVE(1)=IN(J)
      DO 3 K=2,9
```

```
C     STRIP OFF ALPHANUMERIC CHARACTERS
      J=J+1
      IF(IN(J).GT.71B.OR.(IN(J).GT.11B.AND.IN(J).LT.21B).OR.IN(J).EQ.60B
     1.OR.IN(J).EQ.54B.OR.IN(J).EQ.61B.OR.IN(J).EQ.33B.OR.IN(J).EQ.34B.
     2OR.IN(J).EQ.40B)GO TO 4
      ISAVE(K)=IN(J)
    3 CONTINUE
      WRITE(61,103)ISAVE
  103 FORMAT(*  NAME TOO LONG   *,9R1,*. . .*)
      NAME=8H********
      RETURN
C     AT THIS POINT ISAVE CONTAINS THE NAME
    4 ENCODE(8,101,NAME)(ISAVE(K),K=1,8)
  101 FORMAT(8R1)
      RETURN$END

      RESULTS:

      ABC=5.000*XYZ
ABC
XYZ
      A=B+C+D*D/E-F/(G(H+O))
A
B
C
D
D
E
F
G
H
O
      A B C D E= 123
A
B
C
D
E

ABCDEFGHIJK=ABC+DEF
 NAME TOO LONG   ABCDEFGHI . . .
********
JK
ABC
DEF
```

Chapter 7

Problem 7.1(b)

$$x = ((c \uparrow d)/e) + (a * b)$$

Problem 7.2(b)

$$ab{-}c{-}d/e/$$

Problem 7.3(b)

The initial string is

$$(-ab + c)^{-(d-e)}$$

which converts to the reverse polish string:

$$a{\sim}b{*}c{+}de{-}{\sim}{\uparrow}$$

Using rule 7:

$$ab{*}{\sim}c{+}de{-}{\sim}{\uparrow}$$

rule 6:

$$cab{*}{-}de{-}{\sim}{\uparrow}$$

rule 6:

$$cab{*}{-}ed{-}{\uparrow}$$

Converting back to the usual notational form:

$$(c - ab)^{(e-d)}$$

Problem 7.6

```
    PROGRAM PAREN
    DIMENSION INB(66),OPAND(100),OPER(100),LPC(100),RPC(100),OPERH(99)
    TYPE INTEGER OPAND,OPER,PPC,OPERH,TWOOP,HIER,RSUM
  1 READ 100,INB
100 FORMAT(6X,66R1)
    IF(EOF,60)999,998
998 DO 50 I=1,100
 50 LPC(I)=RPC(I)=0
    PRINT 200,INB
200 FORMAT(/10(8H--------),///7X,66R1///)
    N=0
    DO 2 I=1,66
    IF(INB(I).EQ.60B)GO TO 2
    N=N+1
    INB(N)=INB(I)
  2 CONTINUE
    N1=1
    N2=TWOOP=HIER=0
    DO 20 I=1,N
    L=INB(I)
    IF(L.EQ.1R+)3,4
  3 IF(TWOOP.NE.0)20,18
 18 M=2
 19 N1=N1+1
    TWOOP=1
    OPER(N1)=L
    OPERH(N1)=M+HIER
    GO TO 20
  4 IF(L.NE.1R-)GO TO 8
    IF(TWOOP.EQ.0)18,6
  6 N2=N2+1
    OPAND(N2)=0
    GO TO 18
  8 IF(L.NE.1R=)GO TO 5
    M=1
    GO TO 19
  5 IF(L.NE.1R*.AND.L.NE.1R/)GO TO 12
    M=3
    GO TO 19
 12 IF(L.NE.1R$)GO TO 13
    M=4
    GO TO 19
 13 IF(L.NE.1R( )GO TO 14
    HIER=HIER+5
    TWOOP=1
    GO TO 20
 14 IF(L.NE.1R))GO TO 15
    HIER=HIER-5
    TWOOP=0
    GO TO 20
 15 N2=N2+1
    OPAND(N2)=L
    TWOOP=0
 20 CONTINUE
    IF(HIER.EQ.0)GO TO 601
    PRINT 600
600 FORMAT(*      +++++++ UNMATCHED PARENTHESES ++++++*)
    GO TO 1
```

```
601 OPER=1R
    OPERH(1)=0
    N1=N1+1
    OPER(N1)=1R
    OPERH(N1)=0
    I=0
    NPREV=-1
 30 I=I+1
    IF(I.GE.(N1+1))GO TO 42
    IF(OPERH(I).EQ.0)GO TO 3
    IF(NPREV.LT.OPERH(I))40,1
 40 NPREV=OPERH(I)
    K=I
    GO TO 30
 42 IF(NPREV.EQ.-1)71,41
 41 KK=K
    LSUM=RSUM=0
 31 LSUM=LSUM+LPC(K-1)
    RSUM=RSUM+RPC(K)
    IF(LSUM.EQ.RSUM)GO TO 35
    K=K-1
    GO TO 31
 35 LPC(K-1)=LPC(K-1)+1
    K=KK
    LSUM=RSUM=0
 32 LSUM=LSUM+LPC(K)
    RSUM=RSUM+RPC(K+1)
    IF(LSUM.EQ.RSUM)GO TO 36
    K=K+1
    GO TO 32
 36 RPC(K+1)=RPC(K+1)+1
    OPERH(KK)=0
    I=1
    NPREV=-1
    GO TO 30
 71 K=0
    KK=1
 72 K=K+1
    IF(KK.GT.N1)GO TO 80
    I=RPC(KK)
    IF(I.EQ.0)GO TO 73
    DO 74 L=1,I
    INB(K)=1R)
 74 K=K+1
 73 INB(K)=OPER(KK)
    K=K+1
    I=LPC(KK)
    IF(I.EQ.0)GO TO 75
    DO 76 L=1,I
    INB(K)=1R(
 76 K=K+1
 75 IF(KK.EQ.N1)GO TO 80
    IF(OPAND(KK).EQ.0)GO TO 78
    INB(K)=OPAND(KK)
    KK=KK+1
    GO TO 72
 78 K=K-1
    KK=KK+1
    GO TO 72
 80 K=K-1
    PRINT 100,(INB(I),I=1,K)
    GO TO 1
999 CONTINUE
    END
```

Note: Due to the limited character set of FORTRAN a $ sign was used in place of ↑.

```
    RESULTS:
-------------------------------------------------------------------------

    X=A-B/C-D-E$F
    (X=(((A-(B/C))-D)-(E$F)))
-------------------------------------------------------------------------

    A=B+C
    (A=(B+C))
-------------------------------------------------------------------------

    A=-B$C
    (A=(-(B$C)))
-------------------------------------------------------------------------

    X=A$B+C$D-A/B*C$E
    (X=(((A$B)+(C$D))-((A/B)*(C$E))))
-------------------------------------------------------------------------
```

```
    Y=-X-M+A*B/C*N/M*P$E
    (Y=(((-X)-M)+(((((A*B)/C)*N)/M)*(P$E))))
-------------------------------------------------------------------------------
    J-(B-E)))
    ++++++++ UNMATCHED PARENTHESES ++++++
-------------------------------------------------------------------------------
    X=-A$B+C+R*T/A$B+C$N+C
    (X=(((((-(A$B))+C)+((R*T)/(A$B)))+(C$N))+C))
-------------------------------------------------------------------------------
    X=-(-A+B-(-A))
    (X=(-(((-A)+B)-(-A))))
-------------------------------------------------------------------------------
    A=X$(C-D)
    (A=(X$(C-D)))
-------------------------------------------------------------------------------
    M=-C*G+C-N/C$B*(-K)
    (M=(((-(C*G))+C)-((N/(C$B))*(-K))))
```

Chapter 8

Problem 8.1(e)

```
           Reduced String                      Generated Instructions
⊢ n = ~ a ↑ b / ~ a ↑ b ⊣
0   1 5   4   3 15  24  0
      ↑   ↑
      C   H
                                        LDA
                                                 a
                                        RVSG
⊢ n = ACC ↑ b / ~ a ↑ b ⊣
0   1     4   3 15  24  0
          ↑   ↑
          C   H
                                        EXP      b
                                        ST       τ1
⊢ n = τ1 / ~ a ↑ b ⊣
0   1    3 15  24  0
               ↑   ↑
               C   H
                                        LDA      a
                                        EXP      b
```

Reduced String — *Generated Instructions*

```
|- n = τ1 / ~ ACC -|
0    1    3 15    0
            ↑     ↑
            C     H
```

RVSG

```
|- n = τ1 / ACC -|
0    1    3     0
          ↑     ↑
          C     H
```

RDIV τ1

```
|- n = ACC -|
0    1     0
     ↑     ↑
     C     H
```

ST n

Problem 8.2(a)

```
|- ~ a / ~ b + ~ c -|
0  3   14 23 12 23  0
         ↑   ↑
         C   H
```

LDA b

(RVSG held in reserve)

```
|- ~ a / ACC + ~ c -|
0  3   14    12 23  0
       ↑     ↑
       C     H
```

RDIV a

(RVSG still in reserve)

After the pointers are moved, it is found that the RVSG code must be added to the output since the in-hand operator does not become the next operator to take part in the compilation and thus the contents of the accumulator must be stored.

RVSG
ST τ1

```
|- ~ τ1 + ~ c -|
0  3    12 23 0
          ↑   ↑
          C   H
```

LDA c
(RVSG in reserve)

Reduced String	*Generated Instructions*
⊢ ~ τ_1 + ACC ⊣ 0 3 12 0 ↑ C (under 12), ↑ H (under 0)	
	Normally, at this pont, an ADD would be generated, but the RVSG in hand converts this to SUB with an RVSG still in reserve. SUB τ_1
⊢ ~ ACC ⊣ 0 3 0 ↑ C (under 3), ↑ H (under 0)	
	At this point, an RVSG should be generated which will cancel along with that in hand. The compilation is complete.

Problem 8.3(b)

TEA	*TEO*	*Level*	*In Hand*	*Generated Instructions*
x	⊢	0		
a	=	1		
b	~	3		
	−	12	+	LDA b SUB a (RVSG in reserve)
x	⊢	0		
ACC	=	1		
		3	+	(RVSG generated cancels with that in reserve)
x	⊢	0		
τ_1	=	1		ST τ_1
c	+	2		
d	−	12	⊣	LDA d SUB c (RVSG in reserve)
x	⊢	0		
τ_1	=	1		
ACC	+	2	⊣	(ADD converts to SUB, with RVSG in reserve) SUB τ_1
x	⊢	0		
ACC	=	1	⊣	(RVSG is forced out by =) RVSG ST x

Problem 8.4

```
      PROGRAM ASCAN
      DIMENSION TEA(10),TEO(10)
      DIMENSION LEVEL(10),II(2,10)
      DIMENSION OP(9),A(9)
      DATA(II(1,1)=1H  ,0),
     1      (II(1,2)=1H+,2),
     2      (II(1,3)=1H-,2),
     3      (II(1,4)=1H*,3),
     4      (II(1,5)=1H/,3),
     5      (II(1,6)=1HE,5)
      DATA(LEVEL(1)=0)
      INTEGER TEO,OP
    1 READ 100,(OP(M),A(M),M=1,8),OP(9)
  100 FORMAT(8(A1,F8.3),A1)
      IF(EOF,60)19,20
   19 STOP
   20 I=1$K=1
      J=2$L=2
C     CHECK FIRST COLUMN FOR UNARY
      IF(OP(1).EQ.1H )GO TO 2
C     IF PLUS IGNORE
      IF(OP(1).EQ.1H+)GO TO 2
C     MUST BE -
      IF(OP(1).NE.1H-)3,44
    3 PRINT 101
  101 FORMAT(* INVALID UNARY OP*)
      GO TO 1
C     PUT UNARY IN TEO
   44 TEO(J)=OP(1)
      LEVEL(J)=4
    7 J=J+1
    2 TEA(I)=A(K)$I=I+1$K=K+1
C     FIND LEVEL OF NEXT OP
      DO 4 M=1,6
      IF(II(1,M).EQ.OP(L))GO TO 5
    4 CONTINUE
C     HAVE SEARCHED WHOLE TABLE = INVALID OPERATOR
   14 PRINT 102
  102 FORMAT(* INVALID OP*)
      GO TO 1
    5 INHAND=II(2,M)
```

```
C     IF LEVEL OF IN HAND LE LAST OP COMPILE
   15 IF(INHAND.LE.LEVEL(J-1))GO TO 6
C     STORE OPERAND AND OPERATOR IN TABLES
      TEO(J)=II(1,M)
      LEVEL(J)=II(2,M)
      L=L+1
      GO TO 7
C     FORCE CALCULATION
    6 IF(LEVEL(J-1).EQ.0)GO TO 16
      IF(LEVEL(J-1).EQ.4)GO TO 8
      IF(TEO(J-1).EQ.1H+)GO TO 9
      IF(TEO(J-1).EQ.1H-)GO TO 10
      IF(TEO(J-1).EQ.1H*)GO TO 11
      IF(TEO(J-1).EQ.1H/)GO TO 12
      IF(TEO(J-1).EQ.1HE)GO TO 13
      GO TO 14
    8 TEA(I-1)=-TEA(I-1)
   17 J=J-1
      GO TO 15
    9 TEA(I-2)=TEA(I-2)+TEA(I-1)
   18 I=I-1$GO TO 17
   10 TEA (I-2)=TEA(I-2)-TEA(I-1)
      GO TO 18
   11 TEA(I-2)=TEA(I-2)*TEA(I-1)
      GO TO 18
   12 TEA(I-2)=TEA(I-2)/TEA(I-1)
      GO TO 18
   13 TEA(I-2)=TEA(I-2)**TEA(I=1)
      GO TO 18
C     OUTPUT ORIGINAL EXPRESSION AND RESULT
   16 PRINT 103,(OP(N),A(N),N=1,8),TEA(I-1)
  103 FORMAT(///,1X,8(A1,F8.3),1H=,E16,8)
      GO TO 1
      END
```

```
2.000+   2.000    -0.000   -0.000   -0.000   -0.000   -0.000   -0.000=    4.00000000+000

2.000+   2.000    -0.000   -0.000   -0.000   -0.000   -0.000   -0.000=    0.00000000+000

3.142*   3.142/    4.000   -0.000   -0.000   -0.000   -0.000   -0.000=    2.46739693+000

2.000E   3.000    -0.000   -0.000   -0.000   -0.000   -0.000   -0.000=    8.00000000+000

2.000E   3.000-    3.000*   2.000E   3.000-  999.999   -0.000   -0.000=   -1.01599900+003
```

APPENDIX E

A Summary of the Instructions in the Target Language

Each statement in this hypothetical target language has the format:

```
LABEL    OPCODE    ADDRESS,INDEX    COMMENTS
```

where the OPCODE must always be present to define an instruction. When the OPCODE is absent, the assembled word contains the address (or value) portion of the statement only, the OPCODE portion of the assembled word being blank. In this case no index register parameter is permitted. When the OPCODE is defined as a numeric value rather than a symbolic code, that value (converted to the appropriate internal representation) is stored in the OPCODE portion of the assembled word.

An address that is given a negative value in the assembly statement is to be considered an indirect address. If an indirect address is included in a statement together with an influencing index register (the INDEX portion of the statement), the contents of the register are to be added to the address prior to executing the indirect functions. For example, if the word contains the indirect address 13753 and is influenced by the register 3, which contains the value 00217, then the address is to be extracted from the word at location 14070. If this word contains the address 00279, then this is the address that is to be used in conjunction with the OPCODE in the original instruction. That is, if the instruction

```
LDA – 13753,3
```

were to be executed under the above conditions, it would act effectively as

```
LDA   00279
```

This hypothetical machine is assumed to be capable of more than one level of indirect addressing and to contain an infinite number of index registers. In the assembly code statements, indices may not be defined as variables. For the purposes of minimizing the number of labels in each program, a reference to a word may be defined relative to the word containing the instruction being executed. This is achieved by the notation * which symbolizes the address of that instruction. Thus the address *+2 indicates the address of the word two positions toward higher memory from the current address. Similarly, literal addresses (or values) may be defined by preceding the address (or value) by an equal sign. In the context of the program portions described herein, this is used to indicate a value as opposed to an address.

For the purposes of this text, the hypothetical machine contains only one accumulator (given the mnemonic name ACC) though an extension to the accumulator would be needed, in fact, for the storage of the double length result of a multiplication or to store the dividend of a division. However, these features are not essential to the theoretical context of compilation.

The Assembly Code Instructions

In this description the label portion of the statements are omitted since such labels are common to all statements and do not influence the execution of the instructions or the assembly of the definitions. For the purposes of definition the following nomenclature will be utilized:

C(a) The contents of the word at address a.

V(a) The value of the address portion of the instruction.

IR_i Index register i.

In the above definitions, the address portions are assumed to be evaluated after being effected by the index register contents and any indirect addressing.

OPCODE	*ADDRESS, INDEX*	*ACTION*
LDA	a,i	C(a) → ACC
ST	a,i	C(ACC) → a
ENA	a,i	V(a) → ACC
INA	a,i	C(ACC) + V(a) → ACC
ADD	a,i	C(ACC) + C(a) → ACC
SUB	a,i	C(ACC) − C(a) → ACC
RSUB	a,i	C(a) − C(ACC) → ACC
MUL	a,i	C(ACC) * C(a) → ACC
DIV	a,i	C(ACC) / C(a) → ACC
RDIV	a,i	C(a) / C(ACC) → ACC
RVSG		−C(ACC) → ACC The address portion of this instruction is not used, but may be used for other purposes such as the storage of an address which will take part in an indirect addressing scheme.

EXP	a,i	$\mathsf{C(ACC)} \uparrow \mathsf{C}(a) \rightarrow \mathsf{ACC}$
REXP	a,i	$\mathsf{C}(a) \uparrow \mathsf{C(ACC)} \rightarrow \mathsf{ACC}$
LIR	a,i	$\mathsf{C}(a) \rightarrow \mathsf{IR}_i$
		Note that in this case, the index portion of the instruction does not influence the address but merely defines the destination of the contents of that address. However, indirect addressing will be permitted.
NOP		No operation. The address portion of this instruction is not used.

In order to define the remaining instructions, we shall define an internal register of the computer and label it PC (Program Counter). This register always contains the address of the next instruction to be executed. In circumstances where the instructions are to be executed sequentially, this counter is automatically set to the next sequential instruction. In the following instructions the counter will be set to the next sequential instruction unless it is altered within the instruction being executed.

B	a,i	$\mathsf{V}(a) \rightarrow \mathsf{PC}$
BZ	a,i	If $\mathsf{C(ACC)} = 0, \mathsf{V}(a) \rightarrow \mathsf{PC}$
BNZ	a,i	If $\mathsf{C(ACC)} \neq 0, \mathsf{V}(a) \rightarrow \mathsf{PC}$
BP	a,i	If $\mathsf{C(ACC)} > 0, \mathsf{V}(a) \rightarrow \mathsf{PC}$
BN	a,i	If $\mathsf{C(ACC)} < 0, \mathsf{V}(a) \rightarrow \mathsf{PC}$
JMS	a	$\mathsf{PC} \rightarrow a, \mathsf{V}(a) + 1 \rightarrow \mathsf{PC}$
		Note that this instruction has no index portion.

The following two instructions will utilize another internal register which is used to store an address. This will be given the symbol PC1.

MPT	a,i	$\mathsf{PC} \rightarrow \mathsf{PC1}, \mathsf{V}(a) \rightarrow \mathsf{PC}$
BB		$\mathsf{PC1} \rightarrow \mathsf{PC}$
		The address portion of this instruction is not used.

This assembly code contains only one definition (declarative) statement although words may be defined with a blank OPCODE by merely leaving that portion of the assembly statement blank. The statement:

DA *a*

assembles to a word containing the value defined in the address portion of the statement. This value may be defined either as a literal or symbolically by the use of a label. In either case, the value will be right justified in the word that is equivalent to the address portion of an assembled instruction. However, the value may fill the entire word, overlapping into the OPCODE portion.

Index

Note: Where an entry has more than one page reference, the major reference is in sans serif.