ABSTRACT ALGORITHMS
AND A SET-THEORETIC LANGUAGE FOR THEIR EXPRESSION

Preliminary draft, first part

1970-71

J. T. Schwartz
Computer Science Department
Courant Institute of Mathematical Sciences
New York University

PREFACE

The work of which the present manuscript gives first results has its roots in certain musings concerning the relationship between mathematics and programming in which the author has from time to time indulged. On the one hand, programming is concerned with the specification of algorithmic processes in a form ultimately machinable. On the other, mathematics describes some of these same processes, or in some cases merely their results, almost always in a much more succinct form, yet in a form whose precision all will admit. Comparing the two, one gets a very strong even if initially confused impression that programming is somehow more difficult than it should be. Why is this? That is, why must there be so large a gap between a logically precise specification of an object to be constructed, and a programming-language account of a method for its construction? The core of an answer may be given in a single word: efficiency. However, as we shall see, we will want to take this word in a rather different sense than that which ordinarily preoccupies programmers.

More specifically: the implicit dictions used in the language of mathematics, which dictions give this language much of its power, often imply searches over infinite or at any rate very large sets. Programming algorithms realizing these same constructions must of necessity be equivalent procedures devised so as to cut down on the ranges which will be searched to find the objects one is looking for. In this sense, one may say that programming is optimization; and that mathematics is what programming becomes when we forget optimization and program in the manner appropriate for an infinitely fast machine with infinite amounts of memory. It is the mass of optimizations with which it is burdened which makes programming so cumbersome a process; and one must always remember that the sluggishness of this process is the principal obstacle to the development of the computer art.

These reflections suggest that some of the weight of programming be thrown off by passing from the programming dictions ordinarily used to a more highly "mathematicized" language. Our hope to be able to make something of this general idea is raised by the observation that efficiency has two rather different sides. One is mathematical and abstract in character. What intermediate logical constructs must be built as a process proceeds; how large are the sets of logical objects over which searches must be extended during such a process? The other side of efficiency is machine-related, and basically two-fold. First we must ask the question of inner loop efficiency: into what tabular representations can necessary abstract structures be mapped with advantage; once this representation is established, how efficiently can the necessary coded processes be made to effect these tables? Then we must ask a fundamental question related ultimately to the speed chasm which separates electronic from electro-mechanical memories: how large are the data sets with which an algorithm will force us to deal? How can these data sets best be staged between different grades of memory so as to hurry the completion of an algorithmic process? We may remark that machine-related efficiency issues are apt to have as much or more to do with these memory management problems as with problems of inner loop coding, even though most programmers, especially those with an assembly language background and bias, tend to think more of the latter. Our hypothetical mathematicized programming language would practically by definition mask all machine-related efficiency issues almost completely. There is, however, no reason why it should hide these more abstract issues of process design which can easily have a more important bearing on efficiency. Indeed, and this is one of the benefits for which we may hope, it should by masking the former enhance our ability to concentrate on the latter.

The foregoing considerations lead one to suspect that a programming language modeled after an appropriate version of the formal language of mathematics might allow a programming style with some of the succinctness of mathematics, and that this might ultimately enable us to express and to experiment with more complex algorithms than are now within reach. The notion of language that appears here then demands additional clarification. Speaking very generally, a computer language is a set of notations referencing objects and processes, and satisfying all of the following constraints:

i. A formal distinction between well-formed and ill-formed programs exists, and a "syntax checker" capable of administering this distinction can be built.

ii. That class of objects and processes to which well-formed programs refer can be defined rigorously.

iii. A "compiler" capable of transforming a well-formed program into the objects and processes which it represents can be built. These objects can in fact be represented within a computer, and the processes can in fact be carried out.

Since it refers as a matter of course to infinite sets, the language of mathematics has only the first two of these properties, not the third. Nevertheless, it is clear that in searching for a mathematicised programming language we will wish to start from some appropriate version of the language of mathematics. With which of the several variants of formal mathematics which might be contenders shall we begin? It is not unreasonable to begin with set theory, formally represented, let us say, in its von Neumann-Bernays form. This is a language relatively free of artifice, close to the heuristic spirit of informal mathematics, and a formal system with one or another version of which there is a great body of satisfactory experience. In particular, we know that using the very small and simple set of primitives which this language embodies, the whole structure of mathematics, from abstract algebra to complex function theory, can be built up rapidly, intuitively, and in a manner largely free of irritating artificialities. Taking this as our starting point, our problem becomes: adapt set-theory to be machinable.

Not intending to cope with the still largely impenetrable difficulties of mechanical theorem proving, we mean to calculate not with logical formulae but with sets actually enumerated, and this means that our language will only be capable of dealing with finite sets. Given that this is our plan, we must first inquire as to the extent to which the primitive notions embodied in the axioms of set theory are consistent with such a restriction. An easy examination shows that, with the single and obvious exception of the axiom of infinity, which asserts that there exists a set consisting of all the integers, every other basic construction of set theory leads naturally from finite sets only to finite sets. Thus to begin our language we have only to abandon the mathematician's habit of referring to the integers as a closed totality. What replaces such references are the process dictions familiar from other programming languages, that is, dictions which allow (possibly recursive) sequential invocation of a set of basic processes until a desired state is reached. Thus we cook our stew of two sets of ingredients: partly the static dictions of set theory, restricted however to finite sets exclusively; partly of the while's, go-to's and call's from the programming languages. In cooking, these two originally disparate sets of elements will of course interpenetrate each other.

And when it all comes out of the pot, what do we have? To begin with, only yet another programming language, an entity therefore which the sophisticated have increasingly learned to despise. Is it reasonable to expect that the definition and implementation of such a language will, merely because of the mathematicized character of the language, create any advantages at all? I contend that the answer is yes, contend moreover that the benefits which such a language will provide are numerous and substantial, and begin to buttress this claim by indicating one of the most general of the beneficial effects expected, an effect upon computer science education. The availability of a mathematicized programming language should in relatively short order lead to a restructuring of the way that computer science is taught.

Presently one begins with a basic course, titled variously, but generally called something like "Introduction to Data Structures and Algorithms". This fundamental subject matter should now fall into two parts, which might be separately called abstract algorithmics and concrete algorithmics. Abstract algorithmics will be concerned with the depiction and analysis of complex algorithmic processes, independently of the way in which the logical objects to which they refer are to be mapped into a computer. The present work is intended as a first illustration of what abstract algorithmics is ultimately to be. Concrete algorithmics, on the other hand, has the following as its problem: given a family of abstract objects and processes which are to affect them, how can these objects best be mapped into tabular form and the associated processes actually carried out? Note that by isolating and first solving some of the problems of abstract algorithmics, we may expect to be able to discuss concrete algorithmic problems in a more satisfactory manner than has hitherto been typical. Before one knows what in a complex situation one wants to do, one is really not in a very good position to study the ways in which one might do it. Conversely, once an abstract algorithm is put forth, one is generally able to envisage a much wider range of concrete approaches to its optimization than would otherwise be possible. From the point of view of the present work, prior systematic attempts at the depiction of algorithms have generally failed to separate the abstract from the concrete parts of the algorithmic questions which they study, but have mixed the abstract with the concrete, rather to the disadvantage of both. It may also be mentioned that it is abstract rather than concrete algorithmics which stand closest to a third principal branch of computer science, that which plays so large a role in Donald Knuth's magnificent series of books, namely the formal performance analysis of algorithms.

A second benefit is this: the succinctness and descriptive power of a mathematicized programming language will enable us to depict complex processes in their totality, in decisive detail,

and in a form free of abstractly irrelevant specifics. Since our view of complex abstract algorithms will be total and detailed rather than fragmentary and vague we will have a better chance to consider the form and effect of variations in our algorithms, and possible generalizations of them. Bringing the abstract kernel of a process out from behind the veil of abstract irrelevancies which normally obscure it, we will make this kernel more communicable and hence more capable of systematic rational discussion than would otherwise be the case. The limination from formally stated algorithms of irrelevant specifics has still another substantial benefit. Specificity is a major source of incompatability between algorithms, and abstractness will therefore enhance compatability. Consider, for example, the problem of piecing together a compiler out of its customary principal components: parser, optimizer, code generator, etc. We find, typically for a whole class of similar situations, that across each of the interfaces between principal modules some collection, generally rather small, of structured data items must be passed. From the abstract point of view, these will be unproblematical enough: they may be trees, graphs, a few mappings defined thereupon etc.

If one module will naturally produce trees or graphs, and another can conveniently accept such as an input, no serious problem of compatability is to be expected at the abstract level, and separately written modules can readily be fit into a totality. The situation immediately becomes different when abstract data structures are mapped into concrete tabular form. To do so is to define a host of pointer and indexing mechanisms, supplementary variables, overflow conventions and flags, special abbreviations relating to particular data subcases, field sizes, punctuations, and so forth. These definitions, precisely because from the abstract point of view they are largely arbitrary, will never be cast in precisely the same way for two separate program modules except by careful preplanning; and of course, any deviation from perfect agreement may require data restructurings so complex and touchy as to be prohibitive. In this we have the

cause which generally makes it impossible to design useful program library components which will either accept complex data structures as input or provide such structures as output, unless, as is the case with the SETL language to be described in the chapters which follow, a language of sufficient generality provides a fixed framework of conventions. For this reason, intermodule data interfaces often become the foci of major difficulties during the design of large programming systems, wherein months are often consumed in negotiations concerning the detailed layout of data structures to be passed between principal modules. It is normally the case in such situations that all the technical groups involved have elaborated their design ideas not abstractly but in terms of certain implicit assumptions concerning data layout, assumptions which understandably enough they become loath to give up. Moreover, no powerful algorithmic communication technique permitting one group to gain an understanding of the processes which the others are going to apply is available, so that the trade-off issues involved in the choice of data layouts tend to remain obscure on both sides. Naturally, it is then hard to come to intelligent technical compromises. The systematic pre-elaboration of algorithmic strategy using a powerful specification language should cast a welcome light on this dark and perilous corner.

Given that the elimination of irrelevant specifics will restrict the tendency toward incompatability of program modules in a significant way, we may expect to be able to produce fairly complex standard formal algorithms adaptable for use in a variety of situations. Thus it should become possible to put larger items into the cabinet of prefabricated programs than have hitherto filled it. In the present work effort will be devoted to doing just this.

Yet a third anticipated benefit which we expect the use of a mathematicized specification language to provide is this. Our method allows the abstract specification of an algorithmic

process to go forward to completion before any of the concrete table-and-code design issues connected with it have to be faced. When our language is implemented, it will even be possible to execute the abstract programs, and to verify their correctness experimentally. During this process a much smaller mass of program text will be involved than is now the case, it will be much more feasible than it now is to experiment with significant variations in approach during the development of an algorithm. Then, having a debugged algorithm in hand, on will be able to survey it to get a detailed picture of all the data structures which it involves, of all their parts, and of all the processes which must effect these parts. It should then be easier than is now the case to come to sophisticated final decisions concerning the table-structures, data management strategies, and code techniques to be used in a highly efficient version of the same algorithm. In current practice, both classes of issues, abstract design and concrete layout, must be faced at once, and generally in a situation of confusion in which a programmer, already forced to cope with all the complexity which he can juggle, may be unwilling even to contemplate promising optimizations if they threaten to add to the mass of material which he must sustain. Moreover, in typical current situations it is quite hard to maintain design balance, with the consequence that certain parts of the system may be overdesigned, while others, equally or more crucial, may be sorely neglected, and their insufficiency discovered only when it has become impossible, or at least extremely expensive, to do anything about it. The algorithmic language which we propose should, in short, allow the full complex of program-design issues to be approached in orderly stages, and allow minor matters to be classified as such, whereby design attention can be concentrated to ferret out highly effective solutions to key problems.

Most of the benefits to which we have till now alluded come from the use of a suitably powerful specification language, independently of whether this language is implemented. When the mathematicized programming language we project is available for running, however, additional advantages will accrue. A language of this kind cannot but be a most appropriate tool for those situations, especially characteristic of university programming, in which experimental algorithms are developed to be run a few times and then improved or discarded after certain aspects of their behavior are observed. A tool of this kind will also be useful when an elaborate program needs to be built to run just once, or when a complex program whose sole task is to prepare tables for some other program must be produced, or when meta-compilers or other large programs of infrequent use must be prepared, etc. The mathematician desiring to experiment with combinatorial situations, but unwilling to make a very heavy investment in programming, will find a language of the type projected most welcome. The computer scientist will find that it allows him to realize more elaborate algorithms than would otherwise be in reach. Such attractions have made APL increasingly popular; I consider that the partly set-theoretic nature of that ingeniously devised array language lies at the root of this phenomenon.

In the development of large programming systems within an industrial setting, a technique allowing the rapid and inexpensive development of functioning, even if inefficient, versions of complex programs must also be of decided advantage, and this for several reasons. Presently large-program development suffers badly from the fact that little or nothing begins visibly to function until a huge whole has been brought far along. At this point, vast sums may have been spent and time irrecoverably invested. It is then generally the case that what is done is done, and that a project must either bull through along a fixed course, whatever its internal or external deficiencies, or die. Simply by shortening the perilously long feedback loops which characterize present development techniques, an executable specification language would prove of great advantage.

It may also be remarked that during the development of a large system substantial expenses are occasioned by the fact that in such projects it is often necessary to write masses of scaffold-code against which developing systems components can be tested. Intending that our mathematicized specification language should obviate this, we have been at pains to specify an interface linking the specification language to a conventional field-manipulation language of the kind which would normally be used for systems programming. In tandem with such a "lower level" language, our specification language can act as a test-case generator.

The full development of this idea leads to what might be called a two-stage development technique. The first of the two programming stages consists of the development and debugging of a complete systems algorithm, written in the abstract language, and the annotation of this algorithm with all those remarks concerning intended concrete techniques and data management strategy necessary to define the detailed program which is to be developed during the second stage. This first programming stage will also involve measurement and user testing, wherein the abstract algorithm serves as a kind of detailed simulator of the efficient program which it foreshadows, and wherein it may be modified as necessary. During the second stage of programming, all the parts of an abstract code are progressively replaced with logically equivalent but much more efficient passages of concrete code, which are hammered out against the abstract algorithm.

All plans involving execution of abstract specification languages must eventually hope to demonstrate that, even though the efficiency losses which the highly generalized standardization of data representations must occasion will be large, they need not be catastrophic. Loss factors of ten, or even of 100 can be borne; loss factors of 1000 (which we do not at all anticipate) would be disastrous. Technology has after all increased memory capacities by a factor of 100, and speeds by even larger factors, over the last dozen years, and promises

to continue making similarly spectacular gains. Would it not for many purposes be clearly worthwhile to go back a generation in machines, if by doing so we could increase by a large factor our ability to program?

As a final benefit, we expect the availability of the mathematicized programming language which will be described in the present work to broaden the frontier of contact between programming and mathematics. It should at any rate serve to emphasize to the mathematician that programming need not be a mass of petty detail only, that in fact it is concerned in a way only slightly unfamiliar with some of the issues which he is accustomed to confront, that interesting inductive proofs can in fact be regarded as recursive algorithms, and so forth. While the mathematician will presumably find the language which is described in the present work more familiar, and hence more accessible, than customary programming languages might be, the programmer coming to it with a conventional background will find it necessary to change certain of his central habits, and this may at first be rather disconcerting. Conventionally, the mental process of program elaboration which eventually results in a finished program design or program begins not only with half-formulated procedure kernels, but as much as anything else with some idea of the data structures which are to support the procedures to be employed. Often enough of a total design the first part which appears on paper is an initial elaboration of these data structures, their fields, and the separate significances of these fields. This data-depiction is conventionally used as an anvil against which all the detailed processes which eventually will form part of a complete package are shaped. From the present point of view, all this, ingrained habit of the most skillful programmers though it be, is defective, since it indiscriminately confounds the abstract essentials of a process to be described with a host of matters of quite different character. Our procedure will be otherwise, and, bypassing very much of this customary matter, or at the very least making it a postscript to rather

than the start of our specification process, we will deal not with tables, fields, and pointers, but directly with those logical associations, correspondences, and sets which conventional tabular data structures ultimately and indirectly represent. The sudden loss of burden which so radical a simplification implies may at first be somewhat disorienting, and the new medium may at first seem too rarefied to breathe. To the programmer whose initial reaction is as described, the author offers the following advice: don't panic; you are simply emerging from the sea. The necessary new habits of thought are in fact readily acquired, and once mastered will lead to a substantial improvement in one's ability to design eminently practical algorithms. But the necessary design steps will be taken in quite a different order.

It deserves to be made clear that the present manuscript realizes but part of a considerably more extensive plan. Some idea of the intended scope of work may be gained by examining the table of contents which follows, in which we indicate not only sections completed but also sections intended ultimately to follow, not all of which have at the present writing been developed. Of the section titles after IIC shown in the table of contents, II D, III A, III B, and part of VI A presently exist in at least preliminary manuscript form; the others are but projected. Work aimed at implementing the SETL language defined in the following pages is currently under way, and much of the projected but presently missing material will be developed in connection with this project. It should also be noted that, whereas the main outlines of the SETL language now seem reasonably stable, certain features must be expected to change; only wider experience with the use of SETL and a finished implementation will yield a final specification.

Before coming to the end of this preface a few last generalities may appropriately be put to paper. What are programming languages? We would like to suggest the following

outlines of an answer to this questions, evidently fateful for any effort at language design. Programming languages are notational systems devised to facilitate the description of abstract objects whose basic elements are sets, mappings, and processes. With these objects there is to be associated a well-defined rule for evaluating them; perhaps, since the objects may contain processes as subparts, it would be better to say, for interpreting them. From this point of view, an imperative programming language of the ordinary serial kind may be regarded as a mechanism for the description of a set of basic blocks, with each of which is associated a family of possible successors. Each block must also be furnished with a "terminating conditional transfer" which can be used during interpretation of the program to select one potential successor block as the actual "point of transfer". This familiar kind of location-counter control is only one of many quite different possibilities, however. In simulation languages, for example, the basic principle of organization is different: the subprocesses of a simulation naturally form an unordered set, each of which is furnished with an invocation condition. The simulation interpreter executes, in any order, all processes whose invocation condition is satisfied, as long as any remain to be executed; when none remain, an underlying time-parameter is advanced by the interpreter, and the next cycle of simulation begins.

If we bear in mind this broad range of possibilities, the following point of view suggests itself. The "front" or "syntactic" part of a language system must provide methods by means of which very general abstract objects (graph-like, rather than tree-like, i.e., admitting remote rather than purely local connections) can be described conveniently. This "front end" should be variable enough so that the descriptive notation to be used can be tailored to the requirements of any particular field, permitting the objects of most common concern in this field to be described in a succinct and heuristically comfortable manner. Powerful mechanisms for describing the diagnostic or verification tests to be applied

to text during its syntactic analysis should also form part of this language-system front end. The "back" or interpreter part of a language system should incorporate abstract structures which are general enough so that whatsver atructured objects may be of concern in any conceivable situation can conveniently be mapped upon these structures. Now, the use of general set theory should certainly satisfy this latter requirement, as long as the actual use of theorem-proving methods is not at issue. Set theory could only fail to be adequate if some other entities than sets were directly accessible to mathematical intuition and could therefore be used as a fundamental starting point independent of set theory, which is not the case. Thus, if a suitably flexible syntactic front end can be attached to the set-theoretic language with which we shall be working, we will have a system covering a good part of all that is likely to be found along that road which completely bypasses considerations of efficiency. This will in fact be attempted. Of course, this will still leave room for semi-general languages which compromise artfully with full generality in order to reach higher efficiencies than would otherwise be attainable.

Whoever publishes programs publishes bugs; and whereas a certain effort has been devoted to cleaning up the SETL algorithms which appear in what follows, it should be remembered that they have been sight-checked only, not even a syntax-checker being presently available. Indignant readers are asked to send corrections to the author. Readers who would like to receive those parts of the present work which subsequently become available are also asked to communicate directly with the author.

More ample acknowledgement of all those who have assisted, with aid or with criticism, in the development of this manuscript to its present state will be made at a later time. I cannot however, refrain from extending thanks to Mr. David Shields, whose quite effective and unselfish energies have been essential in securing the timely appearance of that which now follows, and to Miss Constance Engle and Miss Linda Adamson, who did such a fine job as typists of the manuscript.

COURANT INSTITUTE OF MATHEMATICAL SCIENCES
JANUARY 1971

Table of Contents

7. Parsing
8. Algorithms for permutations
9. Some additional compiler-related algorithms
10. Table and data compaction algorithms
11. Representation within SETL of other languages
12. Symbolic manipulation: some theorem-prover algorithms

D. Outlines of an implementation

III. Linkage to a lower level language

A. Encode and decode statements

Structures for data layout.

Linking statements.

B. An example of two-stage programming style

C. SETL as a design tool. Concrete algorithms of small compiler front ends.

Features desirable in lower level languages.

IV. Language extensions

A. Introduction. The goals of language extension.

B. Syntax macros.

C. Specific extensions.

a. A SETL based simulation language

b. Extensions for operating system programming.

Description of an operating system.

V. A formal self-description of SETL

A. Front-end: lexical and syntactic data

Dynamic compilation.

B. Target code forms

C. Optimization

a. Intended optimizations

c. Optimization algorithms

D. Lower-level data structures

E. Concrete algorithmic issues

VI. Auxiliary facilities

A. Debugging facilities.

B. An interactive console system.

I. Introduction

In the present work we will propose a new programming language, designated as SETL, whose essential features are taken from the mathematical theory of sets. SETL will have a precisely defined formal syntax as well as a semantic interpretation to be described in detail; thus it will permit one to write programs for compilation and execution. It may be remarked in favor of SETL that the mathematical experience of the past half-century, and especially that gathered by mathematical logicians pursuing foundational studies, reveals the theory of sets to incorporate a very powerful language in terms of which the whole structure of mathematics can rapidly be built up from elementary foundations. By applying SETL to the specification of a number of fairly complex algorithms taken from various parts of compiler theory, we shall see that it inherits these same advantages from the general set theory upon which it is modeled. It may also be noted that, perhaps partly because of its classical familiarity, the mathematical set-notion provides a comfortable framework for thought, that is, one requiring the imposition of relatively few artificial constructions upon the basic skeleton of an analysis. We shall see that SETL inherits this advantage also, so that it will allow us to describe algorithms precisely but with relatively few of those superimposed conventions which make programs artificial, lengthy, and hard to read.

Having general finite sets as its fundamental objects, SETL will be a language of very high level. Generally speaking, we may regard the level of a language as being high to the extent that it succeeds in getting away from the requirement of strict locality of operation which adheres to

an elementary automaton. That is, a high level language is one which incorporates complex structured data objects and global operations upon them. Such a language can free its user from the onerous task, artificial from the abstract structural point of view, of specifying the detailed internal tables which are to represent the structured objects of his concern and of reducing to purely local functions the global transformations which affect these objects. The programmer is thereby freed to write of abstract problem-related entities and their interactions in a familiar and analytically natural manner.

There is, of course, a price to be paid for these very great advantages. A high level language, which reduces to a minimum the amount which a programmer must write to specify an algorithm in executable form, is apt to become committed to the invariable use of certain standard tabular forms for the representation of the entities with which it is concerned, and to the use of certain standard procedures for their manipulation. It will generally use these tables and procedures even in cases in which the nature of a particular process to be programmed allows the use of much more efficient data representations and manipulations. Thus, the use of languages of very high level will lead in many cases to the generation of very inefficient programs. To the extent to which we are unable to capture the optimizing inventiveness of a skilled programmer by an optimization algorithm, this difficulty will persist quite generally. Set theory, which in principle regards a function like cos and the set of all pairs $\langle x, \cos(x) \rangle$ which it generates as being equivalent, certainly tends to the use of dictions implying highly inefficient algorithms, and whereas we will find ways to avoid the worst of these in SETL and will discuss

various ways in which SETL programs can be optimized, it will still be true that SETL will pay a substantial price in efficiency for its logical power.

Nevertheless, it is our feeling that this substantial objection is not catastrophic, i.e., that languages of the type of SETL can be quite useful in a variety of significant situations. SETL will, in the first place, be useful as a specification language, i.e., as a language in which algorithmic processes can be formally and precisely defined by a text whose *syntactic* correctness and completeness may be verified computationally. The value in the definition of highly complex objects of the use of formal text has been emphasized by researchers at the IBM Vienna Laboratory (c.f. Lucas, Lauer, and Stigleitner [1]), who have developed a logical metalanguage called VLD incorporating various powerful set theoretic features and have applied it to give a formal definition of the PL/I programming language; we will make various comparisons between SETL and the Vienna VLD definition language below. It may be remarked that the use of formal text for the definition of algorithms is also a step toward the verification of algorithm correctness and equivalence by formal proof methods; unfortunately, such a step does not bring us very far toward this rather distant goal, as algorithms of the sophistication necessary for such proofs are lacking at the present time.

Once the basic structure of the SETL language has been outlined, we will demonstrate the use of the language by proceeding to describe a variety of algorithms in SETL. Our algorithms will by intent be highly miscellaneous, though, in accordance with the intended application range of SETL, they will all be non-numerical. Specifically, we

will survey as examples algorithms having to do with lists, trees, sorting, parsing, compiling, language definition, optimization, group theory, algebra more generally, combinatorics generally, coding and decoding, simulation, and various other fields. These examples will be intended to demonstrate that the set-theoretic programming technique realized in SETL does allow the description of algorithmic procedures in a brief yet precise and reasonably transparent fashion, and thus to make SETL plausible as a language for algorithm specification. We will see in examining these examples that the use of SETL allows one to evade many of the burdens otherwise associated with programming and to write programs that follow an initial underlying path of thought more closely than is ordinarily possible. In particular, one of the essential initial steps in programming, the design of data structures to carry essential information, is drastically simplified in SETL.

There are other uses than formal definition to which a powerful but rather inefficient programming language like SETL may be put. In certain situations, fairly complex algorithms must be programmed in order to run just once; this is the case, for example, whenever a new language is being "bootstrapped" into existence. In other cases, as for example in experimenting with complex algorithms, in measuring the performance of such algorithms, or in the simulation of complex systems, it may be appropriate to program elaborate procedures to be executed just a few times. In such cases, algorithm inefficiency may be of minor importance.

Still another significant advantage of the availability of a language of very great expressive power can come from

its use in the development of what may be called a *two-stage programming style*. In such a style of programming, algorithms eventually to be incorporated in highly efficient production programs would first be written out in SETL, appearing as short (but executable) texts of a high level of abstraction. Once a verified SETL program was available, its systematic optimization, constituting the second stage of the whole programming process, would begin. This optimization might proceed approximately as follows. Examining a SETL algorithm and noting the sets which played an important role in it as well as the operations to which these sets were subject, one would elaborate a specialized *data description* which could lead to an efficient program for the same algorithm. Note in this connection that sets of certain special kinds appearing frequently in programming situations permit various specially efficient representations. Thus, for example, a subset B of an explicitly represented set A can be represented by a collection of one-bit flags logically attached to the elements of A, and either housed with these elements or segregated into a bit table; a set of ordered pairs may be represented by a two-dimensional array, possibly of bits; an interval of integers may be represented implicitly by a numerical range; etc. Many such representations are known, and constitute much of the normal stuff of programming. To extend such a catalog of techniques, we may note that sets can be represented by arrays, either sorted or unsorted; by lists structured using pointers; by entries in hash tables; or by data structures combining all of these representational methods, as for example structures in which a first few set elements are represented in one way while the remaining elements of the same set are represented in another. Sets of ordered pairs may be represented by attaching all those second components B which occur with a given first component

A as a list; alternatively, all the elements of this list may follow A in a packed array, permitting the representation of a set of ordered pairs in a highly compressed form. Special bit-table techniques are available for the representation of sets fairly dense in a known range; for the representation of functions defined on thin subsets of a known set, etc. Special systems of grouping, advantageous when some part of the total representation of a set is to be stored on an external storage device, are also known.

Since until now one has generally lacked convenient means for the formal description of abstract algorithms, the varied techniques described in the above paragraph, which may collectively be said to constitute the *concrete* part of the theory of algorithms, have more often than not been discussed in intimate connection with that rather different material which we prefer to think of as *abstract algorithmics*. SETL will allow us to deviate from this tradition, and to separate, into two distinct stages, the problem of designing and precisely describing the abstract structure of an algorithm on the one hand, and the different problem of mapping this algorithm efficiently onto a given machine by choice of data structures and procedures coded in a language of lower level on the other. Such separation will hopefully allow us to be more accurate in our treatment of both these two problems than we could normally be if, as is now generally the case, we were constrained to treat them together. It may in particular be noted that a given abstract algorithm can have many plausible concrete images, some of which might be missed if both the abstract and concrete form of an algorithm have to be designed together. We may also hope that the availability of a tool like SETL will enable the accumulation in useful form of "pre-fabricated"

abstract programs. Because abstract programs are free of much of the specific detail which associates itself with algorithms in more concrete form, pre-fabricated abstract algorithms of this kind might serve as permanently useful blueprints for the development of efficient concrete programs. At any rate, we can regard the development and debugging of an abstract algorithm as completing the first stage of a two-stage programming process.

In a formal two-stage programming style the second stage of programming will begin with the choice of a special representation for each of the sets playing a significant role in an algorithm P to be redeveloped into a form more efficient than that directly attained by the SETL compiler. In an industrial setting, the specification of these representational structures, with outlines of the actual procedures by which the set-theoretical processes appearing in P are to be accomplished when the sets appearing in P are replaced by these data structures, might mark the termination of an initial design phase and the passage of responsibility from a design group to a group of final implementors.

The utility of SETL during the remainder of the second stage of programming, i.e., during the elaboration of an efficient final program, can be enhanced by linking SETL explicitly to a more conventional programming language of lower level, which in this introductory discussion we shall designate as the lower language or LL. SETL, being a language of very high level, fails deliberately to provide facilities for the specific layout of data structures, and reflects machine level features useful to efficient programming only in a very indirect way. Internally, it

may represent information in a quite redundant manner. LL, which can be any small "systems programming"-oriented language, that is, any language oriented toward the definition of structured tables and their efficient manipulation, can provide those optimization features which are missing in SETL. In the total language configuration envisaged here and described in more detail below, SETL and LL are both incorporated in a manner allowing them to be intermixed. In particular, once an efficient tabular representation for some structured set S occurring in a pure SETL program has been decided upon, it will be easy, using an appropriate mixture of SETL and LL phases, to describe procedures both for converting S to a tabular representation T and for converting a table having the structure T back to a set structured as S. Such conversion being possible, one may then progressively replace sections of SETL code by expanded versions of the same processes written in LL, converting between tabular and set-theoretic representations for all necessary entities at the points of transition between SETL and LL code. Holding to such a procedure will aid in the orderly elaboration of a final program, and will also provide some assurance that the program as finally developed corresponds to the SETL specification as initially set down. As final LL code is developed, initial SETL statements will progressively be relegated to the status of comments. During this process, those portions of an original SETL code still being executed will serve as "scaffolding" for the developing LL code, providing input data sets and convenient output and checkpoint facilities for testing. Note, however, that as long as a program remains "hybrid", i.e., contains both SETL and LL, it will be inefficient, as the costs of passage of the SETL-LL interface that we shall describe are high.

An alternative technique for the optimization of SETL programs, but one depending on a higher degree of sophistication in the optimization algorithms required, would, instead of using full-blown lower level programming language LL, make use only of a *data structure declaration language* DL, in terms of which any one of an extensive family of describable data structures could be assigned to a set S. Once this were done, it would be the responsibility of the SETL compiler to maintain S in the specified form, and to supply any coded procedures necessary for the application of any SETL operation to S and necessary for the use of S in an environment in which sets maintained in other forms were simultaneously being employed. The provision of such a data structure declaration language would yield a total system easier to use than the SETL-LL system and probably one capable of producing codes of a middling efficiency. We will, in fact, not attempt to persue in any detail the rather ambitious program outlined here, since we judge it to be too ambitious, given the current state of optimisation technique. We confine outselves to giving, in a later section of the present work, an informal survey of various techniques potentially useful for theoptimization of SETL programs. A detailed description both of LL and of a combined SETL-LL language will however be given.

Languages having certain of the features incorporated in SETL have previously been proposed either for general use or particularly for use as specification languages. The APL language developed by Iverson and collaborators (see K. Iverson [1]) has certain set-theoretic aspects, and its use as a specification language has been investigated. Arrays either of numbers or of characters are the basic entity types of this array-oriented language. The size and number

of dimensions of arrays may vary dynamically, allowing arrays to be used in a roughly set-theoretic way, and all operators are systematically extended to arrays, giving a useful type of implicit iteration. The language incorporates selection, location, membership, and concatenation operations. Many set-theoretic procedures can be programmed rather concisely in APL, but the relatively limited subroutine-definition facilities which it incorporates, the fact that subroutines cannot be transmitted as parameters in APL, the fact that regular arrays are in the last analysis considerably more special than general sets of structured data items, and finally that certain essential optimizations are not incorporated into APL cause the current versions of this very interesting language to fall short in expressive power and in certain essential aspects of efficiency when compared to what can be attained by a language explicitly set-theoretical.

The LISP language of John McCarthy and others (see M. Harrison [1]) is of considerable power, and makes use of general list-structured data entities in fully dynamic fashion. In its most basic form, however, it incorporates only a rather primitive syntax, making it rather clumsy for direct use as a specification language. In particular, LISP's insistence that all operators be explicit and be prefixed to their list of arguments, its general lack of implicit conventions and abbreviated modes of expression, and its reluctance to use dummy variables where these could aid the clarity of text, place a crushing burden on attempts to use LISP for the writing of concise and transparent specifications. The SETL language which is to be described could be defined in terms of LISP by the addition of a rather elaborate syntactic front-end greatly enhancing the expressive power of raw LISP, but the amount to be added would be so substantial in relation

to the contribution of the basic LISP processor as to limit the advantage of using a LISP base. Still another consideration intervenes. In our intended implementation SETL will gain rather substantial efficiency advantages from the rather systematic use of hashing techniques. Since LISP does not use these techniques, various of the basic operations of SETL would be quite inefficient if programmed in LISP; the multiplication of the substantial inefficiencies inherent in SETL by these further inefficiencies might lead to a package so slow-running as to be prohibitive. Note in particular that a LISP-based set package would ordinarily determine set membership by serial search over lists which might grow to be quite large, and that a LISP program for determining the equality of two sets whose members may occur in quite different order will normally be rather inefficient.

Besides these two well-known examples, various other proposals incorporating set-theoretic ideas to an interesting degree deserve comment. Codd, Burger, and Lunde [1] contemplate a language, SPEC, having a generally set-theoretical character and intended for the writing of program specifications. Their language includes both unordered and ordered sets, which may have other sets as members. It provides certain selection, numbering, location, insertion, and deletion operators, as well as basic facilities for manipulating character and bit strings; conditional expressions are also provided. On the other hand, no very convenient method for treating a set of ordered pairs as a mapping is included in this language, and the language includes neither labels nor go-to statements, and uses statements of a rather limited "while-condition" form to specify iterations. Perhaps more seriously, no precise formal syntax for SPEC is given, nor is an implementation described; the language thus remains a

semi-formal system for setting down formal comments into which informal remarks may be interspersed, and thus does not seem to attain the full status of a programming language.

A still more elaborate and powerful specification language, designated as ULD, was described by Lucas, Lauer, Stigleitner in [1] and applied by them and their collaborators at the IBM Vienna Laboratory to build up an extensive formal description of PL/I. ULD may be regarded as a version of set theory specialized for use in connection with the tree-like structures arising naturally in connection with language syntax. The basic entities of ULD are tree-structured objects defined by finite sets of nodes and finite collections of transformations ("selectors"), each mapping a node to one of its subnodes. An operation which combines separate trees into a composite tree is provided, as well as an operation which replaces a designated part of a tree T by some other specified tree T; this latter operation serves within the ULD language as a kind of generalized assignment operator. In addition to these structured "objects", the language also provides a class of atomic objects, allows one to speak of integers, and provides comparison operators, boolean and arithmetic operators, as well as universal and existential quantifiers to be used in the formation of boolean expressions, giving an expressive power at least equal to that of the calculus of propositional functions in logic. A choice operator (ix) (p(x)) selecting the unique element x (in an implicit range) having a specified property p(x) is provided, and a set-building operation of the form {x|p(x)}, allowing the construction of the set of all elements with a given property, is provided also. The ULD language seems only to be intended for use as a language of

specification, and has not that rigid degree of formalization necessary make it executable as well. At any rate no entirely formal syntax is specified for ULD, and certainly none which would permit ULD-written programs to be executed. It may also be noted that, while most of the constructions normally used in ULD can be expressed in strictly finitary set-theoretic terms, the language does not hold rigidly to this constraint, but allows quantifiers having infinite sets as their range, as well as expressions involving a variable number N of quantifiers, N itself being existentially quantified; even expressions quantified over some informally understood class of "all predicates" seem to be allowed.

The ULD language, having tree-like objects as basic entities, lends itself quite naturally to the discussion of language syntax. As compared to standard set theory, however, it seems to be of somewhat doubtful real advantage even in this areas, and be distinctly clumsier as a specification tool in other areas. The language includes no labels and no go-to statements. Perhaps more seriously, the manner in which ULD is able to consider a set of pairs as a map is rather artificial: maps between sets of atoms are modeled by allowing atoms b to act as maps, making it possible to regard a map f as a tree node, f(b) then being written as b(f), and thought of as the sub-object which the "selector" b yields when applied to the "node" f.

Its authors use ULD not only to describe the syntax of language, but also to specify those processes of computation which determine language semantics. To do so, the state of a computation is taken to be defined by the momentary state of a tree-like object T, in one distinct sub-tree of which certain nodes are specially marked. These nodes are then

processed in bottom-to-top order, the processing of any node N involving either the replacement of specific sub-trees elsewhere in T by trees built in an elementary way out of the tree branches depending from the node N ("value-returning case"), or involving the replacement of N itself by such a tree. This defines a recursive computational process which ultimately assigns a value to each object T. Lucas, Lauer, and Stigleitner enhance the expressive power of ULD by defining several specialized meta-languages for use in connection with it; one of these serves to abbreviate the description of language syntax in ULD, another provides abbreviation tools for the description of computational processes.

A few comments on standard mathematical set theory itself are worth making. (These comments will be of more interest to those readers with a mathematical background than to those principally interested in programming languages.) An elegant short summary of the axiomatic foundations of the subject appear in Cohen [1], pp. 52-56. Axiomatic set theory uses a very small number of formal primitives, in terms of which the whole structure of mathematics can be built up rapidly and comfortably. We list these set theoretic primitives, and comment on the SETL constructions which correspond to them; of course, whereas mathematical set theory deals with both finite and infinite sets, SETL, intended as an executable programming language, deals with finite sets only. The primitives in question are:

(a) The null set. Provided as a basic entity in SETL.

(b) For two sets a and b, the *unordered pair* {a,b}. Provided as a basic construction in SETL, even in a somewhat generalized form. This construction obviously yields a set (of two elements), thus conforming to the desire of pure

set theory to avoid the introduction of objects other than sets. One next wishes to introduce an *ordered pair* notion, i.e., to define an object <a,b> for each a and b in such a way that a and b can be reconstructed uniquely from a and b. In order that no new object types be introduced, it it is convenient to define the ordered pair <a,b> using the unordered pair construction. One possible definition of <a,b> would be {{a},{∅,{b}}}, but in standard set theory a somewhat different trick is used and one defines the *ordered pair* <a,b> in terms of unordered pairs as follows:

$$\langle a,b\rangle \quad = \quad \{\{a\},\{a,b\}\}$$

Ordered pairs and ordered n-tuples play an important role in a wide variety of set-theoretic instructions. In view of its importance, the ordered n-tuple construction is provided in SETL as a basic operation, and various component modification and construction operations intended to facilitate the use of ordered n-tuples are provided in the language.

(c) For two sets, a and b, their union a∪b. SETL deals with finite sets, permitting the union operation to be defined in terms of a logically more primitive operation which inserts a single additional element into a set. In terms of this operation, and using the function-definition features of SETL, the union operation may be defined trivially.

(d) The "power set" or "set of all sub-sets" construction. This is provided in SETL as a basic operation. For sets as large as the power set of a set which is not very small, efficiency again becomes a serious consideration, and it becomes reasonable to provide as built-in features operations which from the pure mathematical point of view are redundant. As a small concession to this very large fact, we provide in SETL not only the power set construction but an operation

which, for a given set, generates the set of all subsets which have a given cardinality.

(e) Those "choice" functions defined by the various versions of the axiom of choice. In SETL, these are all provided by a simple choice operation ∃x, which in an unspecified but implementation-defined way chooses some element out of each non-null set, x.

(f) The "range of a function" construction. This construction, one of the keystones of set theory in its more elegant representations, asserts that if A(x,y) is any formula of set theory depending on the two free variables x and y, and determining y uniquely for each particular x, then, given any specified set u, there exists a unique set v consisting of all y for which there exists an x in u such that A(x,y). SETL is intended to be executable, and for this reason will not permit a construction quite as general as this; in SETL, we will only allow the range set v to be formed in case we know *a-priori* what set w must be searched in order to find the element y whose existence abstract set theory asserts; or if the element y can be constructed in some explicit way from x using basic and programmer-defined SETL operations, so that we may write y = f(x) using a SETL expression f(x) in which the variable x appears. In the former case, SETL will allow us to write v as

$$v = \{y \in w \mid (\exists x \in u)\ (A(x,y))\}\ ,$$

in the latter case, as

$$v = \{f(x),\ x \in u\}\ .$$

(g) Essentially on grounds of elegance, set theory excludes atoms, i.e., elements other than the null set which have no members. This exclusion, amounting to the insistence that the theory deal only with sets and not with objects of any other kind, is inconvenient in a programming language

context, where one generally wishes for practical reasons to be able to speak of integers and character strings at least, without being forced to the trouble of mapping these additional objects upon objects of pure set theory. Thus for programming purposes a set theory including atoms which themselves have no members but which may freely be members of sets is more convenient than pure set theory. SETL incorporates this convenient modification. Atoms of this kind can be introduced into pure set theory by the following device: choose a set S which is *transitive*, i.e., which includes all the elements of each of its members, and modify the meaning of the basic set-theoretic membership relation x ε y, giving this relation the truth value "false" whenever y is either S itself or is a member of S. This makes S into a collection of atoms, S itself then playing the abstract role of the null set; the original elements of S may be regarded as coded representations of atoms of any other desired nature. Such a construction is of course useful principally to show the consistency within set theory of the assumption of the existence of atoms; in practical terms it is better to give whatever classes of atoms are to be included in the language more efficient special representations of their own, and this is what SETL does.

(h) Set theory assumes at least one infinite set, whose existence is asserted by the normal set-theoretic axiom of infinity. Such a set is used as a starting point to build up the theory of the set of integers (and beyond this, of all of the transfinite cardinals). The sequence of integers is often defined in set theory as ϕ, $\{\phi\}$, $\{\{\phi\}\}$, $\{\{\{\phi\}\}\}$.... etc. A construction of this kind, while set-theoretically very neat, defines the integers using what is in effect a unary representation, which is of course pointlessly inefficient in view of the existence of binary representations for the integers. In SETL, we follow invariable computer

practice and regard the integers as a separate atomic data type, also providing various standard arithmetic operations as basic SETL functions.

In set theory, once the set Z of integers has been defined and the axiom of mathematical induction proved, the existence of Z, and more particularly the possibility of speaking of the whole of Z as a single object, forms the basis upon which one builds all subsequent inductive constructions. To construct a function f(n) defined inductively, the tactic generally used in abstract set theory is as follows: first, using mathematical induction, one proves that for each N∈ Z there exists a unique set of ordered pairs, this set constituting the graph of the function f(n) for all n less than or equal to N; then, using the axiom of range, one forms the union of all these sets of ordered pairs, thus obtaining the graph of the entire function f. In SETL we replace this abstract argument by a construction which not only avoids all reference to the infinite totality of integers but has significant advantages of efficiency, namely by prbgrammed iterative loops and more generally by structured patterns of recursive subroutine calls evaluating an inductively defined function directly. Thus, the label-plus-go-to mechanisms of SETL, and the recursive subroutines which this language provided, are to be regarded as a replacement for the axiom of infinity occurring in abstract set theory. As is known, recursive subroutines alone would suffice for this purpose, but we provide explicitly programmed iterations as well since they furnish a mode of expression more natural in certain circumstances and more efficient in general.

We conclude the present introductory section with a few remarks taken from the general theory of computation which

provide a useful perspective concerning the language which we intend to develop. A computation C is an automatic process by which a specified output pattern OP is to be produced. The computation will begin with some other pattern IP, the program for the computation; will require a certain number of N of time cycles to develop the desired output OP, and will during its activity produce certain masses of intermediate data requiring temporary storage. The size in bits of the initial program IP, the number N of cycles required, and the maximum number S of bits of intermediate storage required during the running of a computation are all highly significant parameters of C. Given OP we wish to minimize all these three parameters; but of course these three different goals will in general not be compatible. The absolutely shortest program specifying a given result is apt to amount simply to a formal description of the result desired, implying a computation, astronomically long in its running time, which searches an entire logical universe looking for an element satisfying the stated criteria. It is in view of the impossible inefficiency of such a procedure that one ordinarily distinguishes between the statement of a problem (concerning finite sets) and an algorithm for its solution.

An algorithmic solution to a problem is by contrast a program which reduces to reasonable limits the size and length of the computation necessary to produce a desired answer. The higher the level of language in which an algorithm is stated, the shorter will be the length of the formal program representing the algorithm; but, with the disappearance of all those optimization hints which might be explicit in a low level language, and with the use of standard rather than specially tailored data representations,

the time and size requirements of the resulting computation will in many cases increase. To the extent that automatic optimizers are able to resupply what is omitted in the high level statement of a problem, part of this degradation in efficiency may be avoided. However, the sophistication of present optimizers is not sufficient to justify much confidence on this score, and certainly optimizers themselves capable of choosing efficient data representations have scarcely been contemplated, let alone designed. Thus, a real price must be paid for the program shortening which a language of very high level brings. A language designer may, according to his intent, choose one of several responses to this dilemma. Maximizing the level and expressive power of his language, he may simply accept the fact that it becomes so inefficient as to be unexecutable in practical terms. In this case, the language will serve only as specification language, in which case its precise formalization serves only to permit the application of whatever static formal syntactic and semantic verification tools may be available. On the other hand a designer definitely intending to produce an executable language may confine himself to the use of statement forms reflecting more or less closely the real properties of the computers which are to execute them, and known by reason or experience to be implementable in a more or less efficient way. In this case, he may produce a FORTRAN, a PL/I, or a SNOBOL. In proposing a new language, we hope to have compromised in an effective way between these two polar tendencies, obtaining a language powerful in its expressions, but still implementable with at least modest efficency. From one general point of view which the foregoing remarks suggest, a programming language is to be judged by the data structures which it incorporates, which should ideally be useful in a wide variety of circumstances

and permit significant efficiencies in their manipulation. In presenting the SETL language which is now to be described we are suggesting that general finite sets can be represented and manipulated in useful ways.

II. Preliminary Account of SETL

II. A. Language Features

In the present section we shall describe the SETL language, omitting various of its features; these additional features will be described in a subsequent section. This will enable us to concentrate on those features of SETL which are newest. The tokens of SETL are names, signed integers, character string constants, and octal constants. Names and signed integers are formed in the usual way; character string constants are included within single quotation marks, and octal constants consist of an octal integer with the suffix B. The formation of character string and octal constants is sufficiently well illustrated by the following examples:

'This is a character string' , 00237B .

SETL makes use of the following special symbols, each of which is a lexical delimiter:

( ) [ ] { } , * + - / = ;

< > Ω ≤ ≥ $ ∃ ∀ ∉ : . | ∋ ∈

Besides the operation symbols appearing in this list, additional operation symbols, both system and programmer defined, are provided, and consist of valid names as in *eq*, *ne*, *with*, *less*, *lesf*, etc. Names may be used as labels, in which case they are followed by a colon.

The basic entities of SETL are *sets* and *atoms*. Sets have elements and are defined by the elements which they contain. Both sets and atoms may be elements of sets. Atoms are either integers, boolean elements, bit-strings, character-strings, labels, blank atoms, subroutines, or functions. Expressions of any of the types set, integer,

boolean, bit-string, character-string, label, blank, subroutine, or function may be written; we shall shortly describe the structure of each of these types of expressions.

a. Set Expressions

If x designates a set or atom, while a designates a set, then x∈a is a boolean expression having the customary meaning. If a is a name designating any set or atom, then {a} designates the set whose only element is a. More generally, {a,b,...,c} designates the set whose only elements are a,b,...,c. We allow a,b,c, etc., to be arbitrary valid expressions, except that, to avoid an ambiguity that would otherwise arise subsequently, we insist that any expression whose central operator is the boolean ∈ be included within parentheses if it is to occur in this construction. Thus, for example, we might write

$$\{(a \in \{a\}), b\} ,$$

this expression would have the same value as the simpler expression

$$\{\underline{t}, b\} ,$$

where t is the boolean atom "true." This bracket construction can be nested to any depth, so that, for example, {a, {a}, {{a}}, {{a},b}} is a valid set-theoretic expression, a,b designating any sets or atoms. The expression

(1) $$\langle a,b \rangle$$

denotes the *ordered pair* of a and b, which, making use of a standard set theoretical device, we will take to be the same as the set

(2) . {{a}, {a,b}} .

(cf. the preceding remarks concerning this construction)

To be sure, our implementation will, in view of the special importance of ordered pairs, treat them using special internal representations; nevertheless, all SETL operations will treat the ordered pair (1) as if it were identical with the set (2). For example, the expressions a ε <a,b> , {a} ε <a,b>, {b} ε <a,b> will have the values false, true, false respectively, except in case a and b are identical, in which case the third statement will have the value true. The ordered-pair bracket construction can be nested to any depth, so that

(3) <<a,b>,c>, <a,<b,c>>, <<a,b>, <c,<d,e>>>

are all valid expressions.

We define the ordered n-tuple <a,b,c,...,d,e> in terms of ordered pairs by the definitional equation

<a,b,c,...,d,e> = <a,<b,<c,...<d,e>>...> ;

so that, for example, the third of the sets (3) may equivalently be written as <<a,b>,c,d,e> . Compound bracket constructions using both pointed and curly brackets may also be written.

Particular elements may be selected from ordered n-tuples by the use of *selection operators*. The simplest of these have the form

<- - ...-*- ... ->

and, as seen, incorporate a number of minus signs, together with a single asterisk, say in the k-th position. This

operator when applied to an n-tuple x, selects the k-th component of x. For example, we have the identities

$$\langle -* \rangle \langle a,b \rangle = b;\ \langle -* \rangle \langle a,b,c \rangle = \langle b,c \rangle\ ;$$
$$\langle *- \rangle \langle \langle a,b \rangle,c \rangle = \langle a,b \rangle\ .$$

If a selection operator containing k symbols (i.e., k-1 minus signs and one asterisk) is applied to an n-tuple $\langle a_1,\ldots,a_n \rangle$, where n>k, we use the fact that $\langle a_1,\ldots,a_n \rangle = \langle a_1,\ldots,a_{k-1},\langle a_k,\ldots,a_n \rangle \rangle$ and regard the n-tuple as a k-tuple. Thus, for example, we have

$$\langle -*- \rangle \langle a,b,c,d \rangle = b, \text{ etc.}$$

Compound selection operators may be written in nested form. For example, we have the identities

$$\langle \langle *- \rangle - \rangle \langle \langle a,b \rangle,c \rangle = a$$
$$\langle \langle -* \rangle - \rangle \langle \langle a,b \rangle,c \rangle = b, \text{ etc.}\ ;$$

whereas in terms of a simple selection operator we would have

$$\langle *- \rangle \langle \langle a,b \rangle,c \rangle = \langle a,b \rangle\ .$$

Note that the operator $\langle \langle *- \rangle - \rangle$ is the same as the composite operator $\langle *- \rangle \langle *- \rangle$, etc. Other selection operator forms, more convenient for use in situations requiring rather elaborate referencing of n-tuple components, are also provided; these will be described in detail in a later paragraph.

The elementary selection operators $\langle *- \rangle$ and $\langle -* \rangle$ may be abbreviated as hd and tl, so that, for example, we have

$$\text{hd } \langle \langle a,b \rangle,c \rangle = \langle a,b \rangle\ .$$

Selection operators may be applied successively, as shown in the following examples:

$$\underline{hd}\ \underline{hd}\ \langle\langle a,b\rangle,c\rangle = a$$
$$\underline{tl}\ \underline{hd}\ \langle\langle a,b\rangle,c\rangle = b, \text{ etc.}$$

The application of a selection operator to an improperly structured set will yield the special atom Ω("undefined") as a result. Thus we have $\langle\text{-}^*\text{->}\langle a,b\rangle = \Omega$ unless b is itself an ordered pair.

The null set is written as $\underline{nl}$.

If f is a set and a any set or atom and if f contains precisely one ordered pair $\langle a,x\rangle$, then f(a) designates the element x, i.e., the *image* of a under f. More generally, f{a} is the set of all x such that $\langle a,x\rangle \in f$. This is the *set of all images* of a by f. If f{a} contains no element or contains more than one element, then f(a) is the undefined atom Ω. Occasionally we will have use for the following still more general construction. If f and a are sets, then f[a] designates the union of all the sets f{x}for $x \in a$. Note then that f{a} may be written as f[{a}].

We allow a number of related, still more general constructions. If f is a set, and g is f{a}, then g{b} may be written as f{a,b}. Thus, f{a,b} is the set of all x such that $\langle a,b,x\rangle \in f$. Inductively, we define $f\{a_1,\ldots,a_n\}$ as $g\{a_n\}$, where g is $f\{a_1,\ldots,a_{n-1}\}$. We define $f(a_1,\ldots,a_n)$ as the unique element of $f\{a_1,\ldots,a_n\}$, if this set has a unique element; and otherwise as Ω. The expression $f\{a_1,a_2,\ldots,a_n\}$ denotes the union of all the sets $f\{x_1,x_2,\ldots,x_n\}$, where $x_1 \in a_1$, $x_2 \in a_2,\ldots,x_n \in a_n$.

It is also convenient to let f{a,[b]} be the union of all the sets f{a,x} for xε b, and more generally let such constructions as

$$f\{a_1,\ldots,[a_i],a_{i+1},\ldots,[a_j,a_{j+1},\ldots,a_n],\ldots\}$$

denote the union of all the sets

$$f\{a_1,\ldots,x_i,a_{i+1},\ldots,x_j,x_{j+1},\ldots,x_n,\ldots\}$$

where $x_i\varepsilon\ a_i$, $x_j\varepsilon\ a_j$, $x_{j+1}\varepsilon\ a_{j+1}$, etc.

In all of these expressions we allow f to be not only a function or set name, but any expression whose value is a function or a set. Thus, for example, the expression

{<a,aa>,<b,bb>,<c,cc>,<a,ax>}{a}

is legitimate, and has the same value as the simpler expression

{aa,ax} .

Of course, if f is a function (which is to say a programmed procedure with a certain definite number of arguments) the correct number of parameters must be supplied to it. Thus, for example, if f is a function of two arguments, the expressions

f(a,b), f[a,b], f{[a],b},

etc., are all valid, while

f(a), f[a], f{a,b,[c]}

are all invalid. (See p. 65 for details.)

If a designates a set, then ɜa designates any arbitrarily chosen element of this set. The notation pow(a) designates the set of all subsets of a, and npow(k,a) the set of all subsets of a consisting of precisely k elements. These operations, while provided in SETL, are good examples of the type of set-theoretic construction which should be used very cautiously if impossibly inefficient algorithms are not to result.

If a is a set and x a set or an atom, then a with x designates the set whose elements are all those of a together with x as an additional element; a less x designates the set whose elements are those of a, excepting x if x is an element of a; and a lesf x designates the set whose elements are those of a, excepting all ordered pairs having x as a first element, if any. Note that {a} with b is {a,b}, that {a,b} less a is {b}, etc.

b. Boolean Expressions

Set equality, inequality, and inclusion are also provided as operators having the form a eq b, a ne b, a incs b. The special symbols t,f denote the boolean values "true" and "false" respectively. The standard boolean operations and, or, not, implies, exor are provided; and, not, and implies may also be abbreviated as a, n, imp. Between two integers m and n the usual comparison operators, having the form eq, ne, ge, le, gt, lt are provided. A set may be tested for being an ordered pair by applying the operator pair a, an an element may be tested for having "atom" status by applying the operator atom a. Bit strings, character strings, labels, blank atoms, subroutines, and functions may be tested

for equality and inequality by using the operators eq and ne.

SETL will use operator precedence rules which are somewhat more restrictive than those which have become customary. These rules may be described as follows.

i. All binary operators, except built-in operators producing boolean from non-boolean values, are to have the same precedence. These latter operators, i.e., the operators (=, eq, ne, inca, ge, le, gt, and lt) have higher precedence, and thus are evaluated first when they occur in compound expressions. Aside from this, unparenthesised expressions containing binary operators of equal precedence are to be evaluated from left to right. Of course, parenthesised expressions will be evaluated in the order indicated by parentheses.

ii. If an operator occurring within an expression is prefixed by the symbol $, the precedence of the operator is lowered one level; this change of precedence applies only to the given occurrence of the operator. If the symbol $ is affixed to the operator, the precedence of the operator is raised one level. Thus, a+b*c+d has the significance ((a+b)*c)+d; the reading a+(b*c)+d can be obtained either by writing a+b*$c+d or by writing a$+b*c+d. Similarly, a+b$*c+d is identical with (a+b)*(c+d).

iii. In much the same way, the precedence of a given occurrence of an operator is raised n levels by the occurrence of n affixed symbols $, and lowered n levels by the occurrence of n prefixed symbols $. Alternatively, n affixed symbols $ may be abbreviated as n$, and n prefixed symbols $ may be abbreviated as $n. Thus, for example, "$3+" indicates that the precedence of "+" is to be lowered three levels. As an example of the use of this convention,

which is intended to prevent the accumulation of parentheses, note that the expression which in conventional notation would appear as (a+b) * ((c+d) * (e+f*g)) * (h+k) could be written using the SETL convention as

a+b * (c+d*e+$f*$g) * h+$k,

or alternatively as

a+b * (c+d$*e+f*$g) * h+$k .

iv. Monadic operators will always have highest precedence, i.e., minimal scope, except as indicated by rule i) or by the occurrence of precedence-modifying "$" symbols. Thus, for example, n x=a is equivalent to n(x=a), while n a and b is equivalent to (n a) and b. The reading n(aandb) can be obtained by writing either $n a and b or n a and$b; similarly, $-a+b represents -(a+b).

v. As indicated below, SETL allows programmer-defined infix binary and monadic operators. The conventions just described apply to those operators also.

If s is a set, and C(x) a boolean formula in which a name x occurs, then a formula of either of the forms

(4) ∃x∈s|C(x)

(5) ∀x∈s|C(x)

represents a boolean value. The value of the first of these expressions, the so-called *existentially quantified* form is obtained by calculating the value of the boolean expression C(x) progressively for each of the elements of the set s, and by assigning the value "true" on first obtaining a "true" result, but assigning the value "false" if no such

result is obtained. The second of these expressions, the so-called *universally quantified* form, is calculated in corresponding fashion but with "false" replacing "true" and vice versa in the preceding description. In forming expressions like (4) or (5) we wish, however, to exclude such ambiguous cases as

$$\forall x \in e \mid (D(x) \text{ or } \exists x \in e_2 \mid C(x))$$

which are in a sense set-theoretical versions of those ambiguous programming sequences in which an iteration variable is explicitly modified within an iteration, or in which an iteration is headed by some such ambiguous statement, for example the FORTRAN statement

```
DO  1  I=1, I+1
```

Such ill-formed cases may be excluded by applying the following technical rule. An occurrence of any name in the role of x in a formula like (4) or (5), i.e., an occurrence of a name x within a part P of a larger formula, P having either the structure $\exists x$ expr Q or $\forall x$ expr Q, is said to be a *dummy*, or *bound*, occurrence of the name x. An occurrence of a name x which is not bound is said to be a *real*, or *free*, occurrence of x. We require, in order that the boolean expressions (4) and (5) be properly formed, that all occurrences of the name x in the boolean expression C(x) be free, i.e., that none of these expressions be bound.

Several additional boolean expression forms closely related to the forms (4) and (5) are provided, and have the appearance

(6A) $\min \leq \exists k \leq \max \mid C(k)$ (6B) $\max \geq \exists k \geq \min \mid C(k)$

(7A) $\min \leq \forall k \leq \max \mid C(k)$ (7B) $\max \geq \forall k \geq \min \mid C(k)$

In these formulae C(k) is a boolean expression in which the name k occurs, all of its occurrences being free; min and max are integer expressions in which k does not occur. The value of the first of these expressions is formed by calculating the value of the boolean expressions C(k) for all integers in the range extending from the value of min to the value of max, assigning the value "true" if C(k) ever assumes the value "true", and assigning the value "false" otherwise. If max < min, (6) has the value "false". The expression (7A) has an equally evident meaning.

The variant forms (6B) and (7B) provide for a variant order of search. Thus, (6A) implies an iterative search in which integers k are tested in *increasing* order until the expression C(k) takes on the value t; and (7A) an iterative search in which integers k are tested in increasing order until the expression C(k) takes on the value f. Similarly, (6B) implies an iterative search in which integers k are tested in *decreasing* order until the expression C(k) takes on the value t; and (7B) an iterative search in which the integers k are tested in decreasing order until the expression C(k) takes on the value f.

Note that the constructions (4), (5), (6), and (7) may be compounded, so that, for example, we may legally write the boolean expression

(8) $\exists x \in e \mid (\min(x) \leq \forall k \leq \max(x) \mid C(x,k))$

Here we require that C(x,k) be a boolean expression in which

all occurrences of x and k are free; that e be a set expression in which neither x nor k occurs; and that min(x) and max(x) be integer expressions which may contain free occurrences of x but which do not contain any occurrences of k.

These restrictions are intended to rule out such ambiguous forms as

$$\exists x \in e \text{ with } x \mid (k-1 \leq k \leq k+1 \mid C(x,k))$$

etc. (cf. the preceding remarks concerning free and bound variables for a more extensive remark on restrictions of this kind).

The following abbreviation of the compound boolean form (8) is allowed:

$$\exists x \in e,\ \min(x) \leq k \leq \max(x) \mid C(x,k)$$

More generally, we allow compound boolean expressions having forms like

$$\exists x_1 \in e_1,\ \forall x_1 \in e_2(x_1),\ \forall x_3 \in e_3(x_1,x_2),\ \ldots \mid C(x_1 \ldots, x_n)$$

which may be taken in an evident way as abbreviations for expressions compounded using the basic construction (4), (5), (6), (7).

In a boolean expression of this kind, we require that e_1 contain no occurrence of the variables $x_1,\ldots x_n$; that $e_2(x_1)$ contain no bound occurrence of x_1 and no occurrence of $x_2,\ldots,x_n$, etc. All but the first of a sequence of like

quantifiers may be omitted, so that, for example, the formula displayed just above may be abbreviated as

$$\exists x_1 \varepsilon e_1,\ \forall x_2 \varepsilon e_x(x_1), x_3 \varepsilon e_3(x_1,x_2) \ldots | C(x_1,\ldots,x_n)$$

A resolving convention is required if the implied scope of a set of quantifiers is not to be ambiguous. One may ask for example, if the expression

$$\forall x \varepsilon e | x \ \underline{\text{gt}}\ 0 \ \underline{\text{or}}\ y \ \underline{\text{eq}}\ 0$$

is to have the reading

$$(\forall x \varepsilon e | x \ \underline{\text{gt}}\ 0) \ \underline{\text{or}}\ (y \ \underline{\text{eq}}\ 0),$$

or the reading

$$\forall x \varepsilon e | (x \ \underline{\text{gt}}\ 0 \ \underline{\text{or}}\ y \ \underline{\text{eq}}\ 0) .$$

We adopt the following convention. A string of quantifiers terminated by a vertical bar is to be regarded as a monadic operator. As such, it will have minimal scope (except insofar as built-in operators producing boolean from non-boolean quantities may have higher precedence). However, the scope of a string of quantifiers may be expanded by prefixing its terminating vertical bar with one or more symbols $. It follows that the first of the above readings is correct; the second may be obtained without the use of parentheses by writing either

$$\forall x \varepsilon e | x \ \underline{\text{gt}}\ 0 \ \underline{\text{or}}\$\ y \ \underline{\text{eq}}\ 0$$

or

$$\forall x \varepsilon e \$ | x \ \underline{\text{gt}}\ 0 \ \underline{\text{or}}\ y \ \underline{\text{eq}}\ 0 .$$

If certain variables $x_1, x_2, \ldots, x_k$ occur in a quantified boolean expression B of the above type, if these variables

are all existentially quantified and are not preceded in the sequence of quantifiers occurring within B by any universally quantified variables, and finally if these variables are enclosed in square brackets, then, after evaluation of the expression B, the variables $x_1,\ldots,x_k$ will take on those first values appearing in the iterative search required to evaluate B which yield the value "true" for B. If no such values exist, then $x_1,\ldots,x_k$ will all take on the undefined value Ω. Thus, for example, if seq is a sequence of sets of integers, evaluation of the expression

min≤ ∃[k]≤max, ∀n∈seq(k)|n gt 0

will cause k to assume the smallest value, in the indicated range, for which all integers in the set seq(k) are strictly positive. Similarly, evaluation of the expression

max≥ ∃[k]≥ min, ∀n∈seq(k)| n gt 0

will cause k to assume the largest value, in the indicated range, for which all integers in the set seq(k) are strictly positive. For a third example, note that evaluation of the expression

max≥ ∃[k]≥ min, ∃[n]∈seq(k)| n gt 0

will cause k to assume the largest value, in the indicated range, for which there exists a positive integer in the set seq(k), and will at the same time cause n to assume a positive value belonging to this set; n will be the first positive integer found in the implementation-defined order of search over the set seq(k). (If no such n exists, n will be assigned the value Ω. The fact that an existential test often implies interest in the element or elements

satisfying the corresponding predicate lends significance to the existential options which have just been described.

c. Set Expressions Once More

We now wish to describe an important type of set expression having a very great expressive power. This *set former* has the following general form:

(9) $\{e(x_1,\ldots,x_n), x_1 \varepsilon e_1, x_2 \varepsilon e_2(x_1), \ldots, x_n \varepsilon e_n(x_1,\ldots,x_{n-1}) \mid C(x_1,\ldots,x_n)\}$

In this general expression, $x_1,\ldots,x_n$ are names; e_1 designates a set-expression not containing any occurrence of these names, $e_2(x_1)$ a set expression not containing any occurrence of $x_2,\ldots,x_n$ and containing only free occurrences of x_1, etc. Moreover, $C(x_1,\ldots,x_n)$ designates a boolean expression containing only free occurrences of $x_1,\ldots,x_n$, and $e(x_1\ldots,x_n)$ designates any arbitrary expression containing only free occurrences of these same names. The notational form (9) is familiar from set theory; its value is the set formed by the following rule: calculate the set e_1; for each of its elements x_1 calculate the set $e_2(x_1)$; for each of these elements calculate the set $e_3(x_1,x_2)$, etc. For each n-tuple $x_1,\ldots,x_n$ obtained in this way and having the property that the boolean expression $C(x_1\ldots,x_n)$ has the value "true", calculate $e(x_1,\ldots,x_n)$; gather all these elements into a set to form the required set. The individual restrictions

(10) $x_j \varepsilon e_j(x_1,\ldots,x_{j-1})$

occurring in (9) may be called *range restrictions*; for use

In certain cases in which ranges of integers play a role, another kind of range restriction, having one of the forms

(11) $\min_j(x_1,\ldots,x_{j-1}) \leq x_j \leq \max_j(x_1,\ldots,x_{j-1})$

is provided. If a range restriction of the type of (11) occurs in the formula (9) in place of one of the range restrictions (10), then, for each appropriate $x_1,\ldots,x_{j-1}$ the two arithmetic expressions $\min_j(x_1,\ldots,x_{j-1})$ and $\max_j(x_1,\ldots,x_{j-1})$ will be calculated, and then in the formation of the set (9) x_j will vary over the numerical range determined by these upper and lower limits (in the indicated order).

If no particular boolean condition C is to be imposed, then the expression (9) may be written as

$$(e(x_1,\ldots,x_n), x_1 \varepsilon e_1, x_2 \varepsilon e_2(x_1), \ldots, x_n \varepsilon e_n(x_1,\ldots,x_{n-1}))$$

The special case

$$(x, x \varepsilon e | C(x))$$

may be abbreviated as

$$(x \varepsilon e | C(x)) .$$

d. Arithmetic Expressions

The arithmetic operators *,+,-,/,// are provided, the last of these operators denoting the remainder after a division. In addition, the built-in arithmetic operators max, min, abs, the first two of these being dyadic, the last monadic, are provided.

If a is a set, then #a denotes the number of elements of a. If s is either a bit string or a character string, then *len* s denotes the length (in bits or characters) of s.

e. String Expressions

All the boolean operations apply on a bit-by-bit basis to bit-strings. If two strings of unequal length are combined by these operations, the shorter is extended by leading zeroes to the length of the longer. Boolean quantities are identified with bit-strings of length 1. Given two strings s and s' of the same type, s+s' designates their concatenation. If n is an integer not exceeding the length of s, the expressions n *first* s and n *last* s designate the n first and the n last bits or characters of s respectively, while n *elt* s denotes the n-th element of the string s and n*s designates the n-fold concatenation of s with itself. The symbols *nul* and *nuls* denote null bit and character strings respectively. Given an integer n, *dec* n and *oct* n designate the representations of n by a character string in the decimal and octal systems respectively.

The allowable characters in a character string are all the normal members of a standard character set, plus one additional character designated *ca* (end record). If s is an ordered pair <st,n> consisting of a character string st and an integer n referencing one of its characters, the system function *record* may be called from within any expression, in the form

record(s) .

The value v of this function is the segment of the string st, beginning at its n-th character, and including all characters of s up to but not including the first occurrence of the character *ca* ; when record is called, it increments the *ff*, n, of its argument by len v + 1. This is an action appropriate for an input reader.

If the n-th character of s is or, then *accord* returns the value *nulc* and increments n by 1. If n exceeds the length of the string s, then *accord* returns the value Ω, and does not increment its argument n; this fact can be used to perform the SETL analog of the normal "end-of-file" test.

Two special strings *input* and *output* are provided. The string *input* is initialized to a value defined by the contents of the system input medium at the time that a SETL program begins to run. Each time additional characters are concatenated to the string *output*, a corresponding printed record will appear on the system output medium. More specifically, each occurrence in the output stream of the character ¢A will terminate the current print line with a period, followed by as many blanks as are necessary to fill out the line. Lines not containing an occurrence of the character ¢A will always be terminated with a blank, followed by a period. This convention allows character strings of arbitrary length to be transmitted to the output medium, and to be represented there unambiguously. If the string *output* is "rewound", i.e., if any portion of it which has already been transmitted to the output medium is modified, a message of the form

OUTPUT MODIFIED FROM CHARACTER n,

n being an appropriate integer, and the print line containing this message being terminated by two periods, will appear.

Printing will then continue, according to the conventions described above, from the first modified character.

In a later paragraph we shall describe conventions which allow the SETL user to indicate that certain character strings are to be stored on external media; by taking advantage of these conventions and by using the operations just described the SETL user will be able to perform all necessary basic I/O.

f. Label Expressions

No operations combining labels, except the equality and inequality comparisons, exist; however, labels may be members of sets and may therefore appear within ordered pairs, so that the result of applying a function to an element may be a label. An expression producing a label value may appear in a *go to* statement (see below). This possibility can be used to obtain a "calculated" *go to* effect in SETL programs.

g. Blank Atoms

No operations combining blank atoms, except the equality and inequality comparisons exist; however, blank atoms may be elements of sets, just as atoms of any other kind. Blank atoms are created by the built-in SETL function newat, which creates a new blank atom each time it is called. Note, for example, that such an expression as <newat, newat> designates an ordered pair consisting of *two distinct* blank atoms. Blank atoms are provided to serve as structural markers in

complex sets built up during the course of a SETL computation, and to facilitate garbage collection.

h. Conditional Expressions

A conditional expression like that of ALGOL is provided; this has the form

if $bool_1$ then $expr_1$ else if $bool_2$ then $expr_2$...else $expr_n$

and has its customary and evident meaning. Here, $bool_1$..., $bool_{n-1}$ are required to be boolean expressions, while $expr_1$,...,$expr_n$ are expressions having arbitrary values.

As an illustration, which the reader may wish to ponder, of the compound use of various of the basic SETL constructions, and especially of the power of the general expression (9), we give an expression solving the following programming problem: given a set seq of ordered pairs $\langle n, x_n \rangle$ representing a sequence, and given any additional element y, express the result of inserting y as the m-th element of the sequence. The required expression is

{<if hd p lt m then hd p $else hd p+1, tl p>, p∈seq}with<m,y> .

A resolving convention is required if the implied scope of the concluding else in a conditional expression is not to be ambiguous. One may ask, for example, if the expression

(12) if x gt 0 then y else x+y

is to have the reading

(13) if x gt 0 then y else (x+y)

or the reading

(14) (if x gt 0 then y else x) + y .

We adopt the following convention. The concluding *else* in a conditional expression is to be regarded as a monadic operator. As such, it will have minimal scope (except insofar as built-in operators producing boolean from non-boolean quantities may have higher precedence). However, the scope of the *else* may be expanded by prefixing the keyword "else" with one or more symbols $. It follows that the normal reading of (12) is (14); the reading (13) may be obtained without the use of parentheses by writing either

if x gt 0 then y else x+$y

or

if x gt 0 then y $else x+y.

We also allow the use of parentheses, in an evident way, to determine scopes in conditional expressions.

i. The Use of Functions Within Expressions; Programmer-Defined Operations; Expressions in General

SETL allows subroutines and functions to be defined in a manner to be described in more detail below. Functions may be used as parts of expressions; a function invocation occurring within an expression will have the form

name (expr1,..., exprn) .

Here, name is the function name, while expr 1,...,expr n may

be arbitrary SETL expressions. These expressions are evaluated before the function is invoked, and the values thereby obtained define the arguments to be transmitted to the function. A function called from within an expression may modify various of the function's arguments, and its invocation may in general cause other "side effects". Function arguments whose values are to be modified should, of course, be simple names rather than compound expressions.

SETL allows programmer-defined functions and subroutines of two arguments to be written as infix operators, provided that this notational intent is appropriately stated in the definition of the function (additional details are given below). Similarly, functions of a single argument may be written as prefixed monadic operators. The name of a function to be written in such an operator form, whether monadic or dyadic, is underlined. Thus, for example, if union and intersect are the names of programmer-defined dyadic operators we may write

(seta union setb) intersect setc

as part of a compound expression.

The basic SETL expression forms may be nested in arbitrary fashion to produce complex expressions. Thus, set-forming constructions may be nested within conditional expressions, which may in turn be nested within n-tuple forming expressions, etc. The order of expression evaluation is inside-out and left-to-right. This observation concerning evaluation order may be important if functions producing side effects are invoked during the evaluation of an expression.

3. Statement Forms. Assignments, Multiple Assignments, and Indexed Assignments

Having said sufficiently much concerning expressions, we go on to discuss the statement forms proper provided by SETL, beginning with a discussion of the *assignment* statement. Single statements of SETL are punctuated with terminating semi-colons. The simplest kind of assignment statement has the familiar form

name = expr;

here *name* must be any valid variable name, while *expr* can be any valid expression. A more general type of *multiple assignment statement* intended to be of convenient use in connection with ordered n-tuples, is also provided. Such a statement makes use of a structured *assignment bracket*. A simple assignment bracket has a form like

$$<---\ldots-x_1-\ldots-x_2-\ldots-x_k--->$$

i.e., consists of n symbols enclosed within pointed brackets, and separated where necessary by commas; of these symbols, a certain number k will be distinct variable names, and the remaining n-k will be minus signs. When such an operator is applied to an ordered n-tuple the k elements of the n-tuple which correspond in position to the names appearing within the assignment bracket are located and each name is assigned the value which corresponds to it in position. If no element corresponds in position to a name x, then x is assigned the special atom Ω as its value. Thus, for example, the multiple assignment

$$\langle x\text{-}y\rangle = \langle\langle a,b\rangle,c,d,e\rangle \quad ;$$

leads to the individual value assignments

$$x = \langle a,b\rangle \; ; \; y = \langle d,e\rangle \; ;$$

the multiple assignment

$$\langle -x\text{-}y\rangle = \langle a,\langle b,c\rangle,c\rangle \quad ;$$

leads to the individual value assignments

$$x = \langle b,c\rangle \; ; \; y = \Omega \quad ;$$

if c itself is not an ordered pair.

If one of the names appearing within a generalised assignment bracket is replaced by an asterisk, the bracket becomes an *assignment operator*, retaining its "assignment" significance with regard to the remaining names, but at the same time becoming a valid component of a set expression, having as its value that particular component of the n-tuple which is located by the asterisk. Thus, for example, each of the statements

$$\langle x\text{-}y\rangle\langle a,b,c\rangle;\; x=\langle *\text{-}y\rangle\langle a,b,c\rangle;\; y=\langle x\text{-}*\rangle\langle a,b,c\rangle$$

is equivalent to the pair x=a; y=c; of individual assignments. (cf. the discussion of selection operators given in a preceding paragraph.) Pointed brackets may be nested to produce somewhat more highly structured generalized multiple assignment brackets and assignment operators. These assignment brackets and assignment operators are intended

to be of convenient use in situations requiring elaborate references to n-tuple components. They will be described in detail in a later paragraph, which will also give details concerning the general forms of selection operators provided in SETL.

SETL also makes use of various assignment forms related to the "indexed" assignment operations conventionally used in programming languages. The simplest such operator has the form

name (expr1) = expr2;

here, *name* must be a valid name while *expr1* and *expr2* must be valid expressions. The value of *name* must be a set. This statement has the same effect as the more complex statement

name = (name less expr1) with <expr1, expr2>

A related assignment form is

name {expr1} = expr2;

here, once more, *name* must be a valid name, while *expr1* and *expr2* must be valid expressions; the value of *expr2* must in this case also be a set. When executed, this statement at first removes all ordered pairs of the form <expr1, y> from the set designated by *name* and then unites the set

{<expr1, z> , z: expr2}

to the result.

The special case f(a)= nl removes all pairs of the form <a,x> from f, and is thus synonymous with f=f less a. We allow the special assignment f(a)=Ω, which has no other reasonable meaning, to have the same meaning as f(a)= nl.

We also permit multiple indexed assignments using assignment brackets. Thus

$$\langle f(a),\ g(b)\rangle = \langle x,y\rangle$$

has precisely the same effect as the two individual indexed assignments

$$f(a) = x;\ g(b) = y \quad .$$

Finally, we allow multiply indexed assignments, having the form

$$f(expr_1,\ expr_2,\ \ldots,\ expr_n) = expr$$

and

$$f\{expr_1,\ expr_2,\ \ldots,\ expr_n\} = expr \quad .$$

The first of these assignments statements is synonymous with

$$f(expr\ 1,\ expr\ 2,\ \ldots,\ expr_n) = \{expr\} \quad ;$$

the second statement removes all ordered (n+1)-tuples having the form <expr 1, expr 2, ..., $expr_n$, x> from f, and then unites the set

$$\{\langle expr\ 1,\ expr\ 2,\ \ldots,expr_n,x\rangle,\ x\in expr\}$$

to the result. The statement

$$f(expr\ 1,\ldots,expr_n) = \Omega$$

is taken to be synonymous with

$$f(expr\ 1,\ldots,expr_n) = \underline{nl}.$$

Multiply-indexed, multiple assignment statements such as

$$\langle f(a,b),\ g(a,b)\rangle = \langle x,y\rangle$$

are also permitted, and have an evident meaning. Multiply-indexed names may also appear in structured assignment operators; so that, for example,

$$\langle f(a,b),^{a},g(a,b)\rangle x$$

is a valid expression whose value is $\underline{hd}\ \underline{tl}\ x$ and whose evaluation causes the assignments

$$f(a,b) = \underline{hd}\,x;\ g(a,b) = \underline{tl}\ \underline{tl}\ x$$

to be made.

These assignment side effects must be carefully noted, and in examining expressions containing such operators, it may be necessary to keep the SETL rules concerning order of sub-expression evaluation in mind. It will not infrequently be possible to make constructive use of the side effects of operators invoked during function evaluation.

k. Statement Forms. Control Statements, Iterations, and Compound Operators.

Control of program flow is provided in several forms; by *go to* statements, by *if* statements, by a statement form specifying iteration over a range of sets, by a statement form specifying iteration for as long as a certain boolean condition is valid; and by subroutine or function calls.

A SETL statement may be labeled by prefixing it with a name, which should be followed by a colon, and which for the sake of readibility may be enclosed in square brackets. The affixed colon designates the name as a label. Thus

label: , [label:] , [[label:]]

are all equivalent valid labels.

A *go to* statement has the form

go to expr;

the expression occurring in such a statement may be perfectly general, but must have a label as its value. An *if* statement may have the form

if $bool_1$ then $block_1$ else if $bool_2$ then $block_2$...else $block_n$;

or may have the slightly simpler form

if $bool_1$ then $block_1$ else if $bool_2$ then $block_2$ ...
else if $bool_{n-1}$ then $block_{n-1}$;

Here $bool_1,\ldots,bool_{n-1}$ are required to be boolean expressions; each of $block_1,\ldots,block_n$ is an arbitrary sequence of valid SETL statements, which may include go to statements and additional if statements.

Each statement block forming part of an if-statement, with the exception of the last such statement block, is terminated by the occurrence of the next following keyword *else* or *then*. The last block is terminated by the occurrence of the semicolon explicitly shown as terminating the if-statement displayed above. Of course, since the final statement of this final block is itself terminated by a semicolon, the visible sign of an if-statement termination will be a double semicolon. For example, we might have

```
if x gt 0 then set = set with x;; else n=n+1;;
```

This style is acceptable for short blocks, but when long blocks of statements are encompassed within if-statements scopes delimited in this way would become confusing. For this reason, we permit a variety of alternative conventions. These are:

1. The use of parentheses. Any block of statements may be enclosed in parentheses, and remains a block of statements. Thus, we might write

```
if x gt 0 then set=set with x;; else (n=n+1;k=k*n;);
```

2. The use of either of the more visible terminators "end if;" and "end else;". Thus, we might alternatively write either

```
if x gt 0 then set=set with x; else n=n+1;k=k*n;end if;
```

or

```
if x gt 0 then set=set with x; else n=n+1;k=k*n;end else;
```

To provide a yet more explicit scope termination, we allow the terminators "end if" and "end else" to be extended by inclusion of the first token following the corresponding keyword "if" or "else" in the statement being terminated. For example, the statements shown above may also be written as

```
if x gt 0 then set=set with x; else n=n+1;k=k*n; end if x;
```

or

```
if x gt 0 then set=set with x; else n=n+1;k=k*n;end else n;
```

3. The use of an explicit indication of scope. This employs the following convention: *til* is a keyword; if the combination

```
til name
```

occurs immediately following the last else of an if-statement, then the block of statements will be terminated not by a semicolon, but by the first following occurrence of the name *name* as a label. For example, we may write the above if-statement optionally as

```
if x gt 0 then set=set with x; else til done; n=n+1;k=k*n;done:...
```

4. The use of a do-block. Macro-blocks of a simple,

easy to use form are provided in SETL. These are called do-blocks, and details concerning them are given below. An over-lengthy sequence of statements encompassed within an if-statement may be replaced by a do-block invocation. This can lead to a more readable program form in which lengthy clauses are removed to an out-of-line position. Using this possibility, we might in the situation contemplated above write

if x gt 0 then set=set with x; else do twostatements;;

where twostatements is a do-block having the detailed form indicated below but logically equivalent to the two separate statements

n=n+1; k=k**n .

Additional details concerning the form and use of do-blocks will be found below.

SETL allows iterations to be specified without the explicit use of labels. Two statement forms serving this end are provided. The first which may be called the set-theoretic iteration has the general appearance

(2) (∀x_1 ∈ e_1, x_2 ∈ $e_2(x_1)$, ..., x_n ∈ $e_n(x_1,\ldots,x_n)$ | $C(x_1,\ldots,x_n)$)block;

In this general expression, $x_1,\ldots x_n$ are names, e_1 designates a set-expression not containing any occurrence of these names, $e_2(x_1)$ a set expression not containing any occurrence of $x_2,\ldots,x_n$ and containing only free occurrences of x_1, etc. Moreover, $C(x_1,\ldots,x_n)$ designates a boolean expression containing only free occurrences of $x_1,\ldots,x_n$, while block

is any sequence S of valid SETL statements and may include go to statements. The statement (2) is executed according to the following rule: calculate the set e_1; for each of its elements x_1 calculate the set $e_2(x_1)$; for each of these elements calcualte the set $e_2(x_1,x_2)$, etc. For each n-tuple $x_1,\ldots,x_n$ obtained in this way and having the property that the boolean expression $C(x_1,\ldots,x_n)$ has the expression *true*, perform the statements of the sequence S in order. (Note that the occurrence in S of a go to statement with a destination label outside S will terminate the iteration implied by the statement (2).)

The individual restrictions

$$(3)\quad x_j \text{ in } e_j(x_1,\ldots,x_{j-1})$$

occurring in (2) may be called *range restrictions*; for use when iteration over a range for integers is desired, a restriction having the variant form

$$(4A)\quad \min_j(x_1,\ldots,x_{j-1}) \leq x_j \leq \max_j(x_1,\ldots,x_{j-1})$$

is provided. If (4A) occurs in an iteration (2) instead of (3), then for each appropriate $x_1,\ldots,x_{j-1}$ the two arithmetic expressions $\min(x_1,\ldots,x_{j-1})$ and $\max_j(x_1,\ldots,x_n)$ will be calculated, and iteration will be extended suitably over all x_j in the numerical range defined by these upper and lower limits.

Numerical range restrictions in the variant forms

$$(4B)\quad \max_j(x_1,\ldots,x_{j-1}) \geq x_j \geq \min_j(x_1,\ldots,x_{j-1})$$

(4C) $\min_j(x_1,\ldots,x_{j-1}) \leq x_j \leq \max_j(x_1,\ldots,x_{j-1})$, etc.

are also allowed. These variant forms provide for variant orders of iteration. Thus, (4A) implies an iteration in which successive integers x_j are treated in increasing order; (4B) implies an iteration in which successive integers x_j are treated in decreasing order.

Iteration over an empty set is allowed, in which case the block of statements in the scope of the iteration is not executed.

The scope of a set-theoretic iteration of the form (2) may be indicated, in the manner shown, by the location of a semicolon otherwise absent. For readibility, however, several alternative forms are provided. These are:

1. The use of a terminator

end Vx;

to close the scope of an iteration which begins

(Vx∈expn, y∈....)

2. The use either of parentheses, of the *till* option, or of a do-block, as described above in connection with the *if*-statement.

A second type of iteration, which may be called the *while*-iteration, is provided in two related forms, of which the simplest is

(5) (while C) block;

Here C is any boolean expression, while *block* designates any sequence S of statements. This statement has precisely the same significance as the following more explicit group of statements

(6) L1: if n C then go to L2;; block; go to L1; L2: ;

where L1 and L2 are unique generated labels.

The scope of a while-iteration may be indicated in the manner shown by the location of a semicolon otherwise absent. For readibility, however, several alternative forms are provided. These are

1. The use of the more visible terminator

end while;

This terminator may optionally be made more explicit by extending it to include the first token following the keyword *while* which opens the iteration

(while x≤s) k=k+g(x); x=f(x); end while x;

2. The use either of parentheses, of the *til* option, or of a do-block, as described above in connection with the *if*-statement.

A rather more general form of the while-iteration is as follows:

(while C doing blocka) block b;

Here C is any boolean expression, while blocka and blockb are arbitrary sequences of statements. This iteration has

precisely the same significance as the following explicit sequence of statements:

(7) L1: if n C then go to L2; blockb; blocka; go to L1; L2:;

where L1 and L2 are unique generated labels. This alternative form of the *while* iteration-header is provided to improve readability by making it possible to attach loop-associated bookkeeping instructions directly to the header, rather than requiring these instructions to be placed remotely.

The instruction

quit;

occurring either within a set-theoretical iteration or within a while-iteration is equivalent to

go to L;

where L is a unique generated label occurring at a position immediately following the last statement in the scope of the iteration. We also allow this statement in several more explicit forms. The form

quit Vx;

is equivalent to (8) where L is a unique generated label occurring at a position immediately following the last statement in the scope of the set-theoretical iteration whose header begins either with

(Vx ε ....)

or with

```
(min≤ ∀x≤ min , .......    )
```

or with some other allowed form of range restriction in which "∀x" appears. Note that this generalized *quit* statement may cause control to be transferred out of several nested iteration scopes all at once. For example,

```
(∀x∈a) y← x; (while ↓y gt 2)(∀z∈y) n←n with z;
    if ↓n gt 10 then quit ∀x;;end ∀z; end while; end ∀x;
```

is equivalent to

```
(∀x∈a) y← x; (while ↓y gt 2)(∀z∈y) n←n with z;
    if ↓n gt 10 then go to L;;end ∀z;;;[L:]...
```

We also permit the forms

```
quit while;
```

and

```
quit while token;
```

The first of these is equivalent to a transfer to a label occurring at a position immediately following the last statement in the scope of the innermost while-iteration containing the quit statement. The second, more explicit, *quit* statement has a significance which may be defined as follows. Let W be the innermost while-iterating header whose scope contains the *quit* statement and which begins with the sequence

(while token .....)

of symbols. Let L be a unique label occurring at a position immediately following the last statement in the scope of W. Then (9) is equivalent to (8).

The instruction

(10) continue;

is used in a closely related manner. If this instruction occurs either within a set-theoretical or a simple while iteration, it is equivalent to (8), where L is a unique generated label occurring within, but at the very end of, the scope of the iteration. Thus, for example,

```
(while x gt 0) x=x-f(x); if g(x) lt 0 then continue;
    else y=y+g(x);; end while;
```

is equivalent to

```
(while x gt 0) x=x-f(x); if g(x) lt 0 then go to L;
    else y=y+g(x);; [L:]; end while;
```

Suppose next that a simple *continue* statement (10) occurs within a *while*-iteration whose header is of the more complex form

(while C doing blocks)

Then, by definition, (10) causes a transfer to a unique generated label located immediately before the various statements of the block *blocks*, which (cf. (7)) form a group terminating the scope of the while-iteration. For example,

```
(while x∈s doing x=f(x);) if g(x) lt 0 then continue;
    else y=y+g(x);; end while;
```

is equivalent to

```
(while x∈s) if g(x) lt 0 then go to L;
    else y=y+g(x);;[L:] x=f(x); end while;
```

We also allow more general *continue* statements having the form

```
(11)        continue Vx;
(12)        continue while;
(13)        continue while token;
```

This is how we define the significance of the statement (11). Let W be the innermost iteration header whose scope contains the continue statement (11) and which begins either with the sequence

```
(Vx∈ .....        )
```

or with some such sequence as

```
(min≤ Vx≤ max, ...        ).
```

Let L be a unique label occurring within the scope of this iteration, but at the very end of this scope. Then (11) is equivalent to

```
go to L;
```

The significance of the statements (12) and (13) may be

defined in similar fashion, and we leave it to the reader to supply the necessary details. Observe, however, that if the while-iteration header to which (12) or (13) refers contains a *doing* block, then (12) or (13) will cause a transfer to a label located immediately before the first statement in this block; note (cf. (7)) that this block of statements occurs at the very end of the scope of the *while* iteration.

We now describe a type of *compound operator* provided in SETL and related to the set-theoretic iterations with which we have just been concerned. If *op* is any binary operator or function of two variables, or more generally any expression having such an operator or function as its value, then

(14) $\{\underline{op}: x_1 \in e_1 \mid x_2 \in e_2(x_1), \ldots, x_n \in e_n(x_1, \ldots, x_{n-1}) \mid C(x_1 \ldots x_n)\}expr$

denotes the value v which would result from the following iterative calculation

(15) v=Ω; times=0; (∀$x_1 \in e_1, x_2 \in e_2(x_1), \ldots, x_n \in e_n(x_1, \ldots, x_{n-1}) \mid C(x_1 \ldots x_n)$)
if times eq 0 then times=1; v=if times eq 0 then expr
else v op expr; if v eq Ω then quit ∀x_1; end ∀x_1;

The "operator" appearing within brackets in (14) is called a compound operator. The construction (14) is subject to the same conditions concerning free occurrences of variables, etc., as is the iteration (2). In particular, numerical range restrictions of any of the forms (4A), (4B), (4C) are allowed. The construction (14) will ordinarily be used when the operator op is commutative and associative, in which case the value of (14) is independent of the order in which the set-theoretic iteration (15) is carried out. Note, for

example, that the availability of the construction (14) allows us to write the ordinary mathematical

$$\sum_{x_1 \varepsilon a_1} \quad \max_{x_2 \varepsilon a_2} \quad \left| \quad \bigcup_{x_3 \varepsilon a_3} \quad s\,(x_1, x_2, x_3) \right|$$

as

$$[+ : x_1 \varepsilon a_1]\,[\underline{\max} : x_2 \varepsilon a_2]\ \$\ [\underline{\cup} : x_3 \varepsilon a_3]\ s(x_1, x_2, x_3).$$

A resolving convention is required if the implied scope of a compound operator is not to be ambiguous. One may ask, for example, if the expression

$$[+ : x \varepsilon a_1]\ f(x) + b$$

is to have the reading

$$([+ : x \varepsilon a_1]\ f(x) + b$$

or the reading

$$[+ : x \varepsilon a_1](f(x) + b).$$

We adopt the convention that a compound operator is to be treated as a monadic operator; as such, it will have minimal scope (except insofar as built-in operators producing boolean from non-boolean quantities may have higher precedence). Thus, the first of the two possible readings noted above is correct. The scope of a compound operator may, however, be varied by the use of the "$" symbol in the manner described in an earlier paragraph, so that we may obtain the second reading displayed above by writing

either

$$\$\ [+\ :\ x \in a_1]\ f(x) + b$$

or

$$[+\ :\ x \in a_1]\ f(x) + b$$

1. Subroutine and Function Definitions. Name Scopes.

SETL provides various features intended for the definition of subroutines, functions, and macros called in various ways. A subroutine definition has the form

```
define name (arg1, arg2,...,argn); body; end name;
```

Here *name*, *arg1*,...,*argn* are all valid names, while *body* is any valid sequence of SETL statements. The final statement

```
end name;
```

in the above definition locates its end. A subroutine defined in this conventional way is called via a statement

```
name (expr1,...,exprn);
```

in which *expr1*,...,*exprn* are expressions defining the actual subroutine arguments to be used at the moment of call. Return from a called subroutine is accomplished using a *return* statement having the familiar form

```
return ;
```

subroutines are always recursive, as are functions (see below).

Note that when a subroutine is called recursively only the subroutine arguments and the variables local to the subroutine are stacked; variables external to the subroutine but known within it (c.f. the discussion of name scoping and the *external* declaration which follows) are not stacked. Thus, for example, a subroutine may be used recursively to add elements to a single particular external set.

Functions rather than simple subroutines may be defined by writing

```
definef name (arg1, arg2,...,argn); body; end name;
```

Return statements occurring within the body of a function definition must have the slightly expanded form

```
return expr;
```

here *expr* is an expression which, evaluated immediately before a return from the function, defines the function value.

In addition to these conventional forms of function and subroutine definition, SETL allows the definition of functions and subroutines called in a manner more closely resembling the normal SETL use of infix and prefix operators. The definition of a function to be called as an infix operator is written as follows:

(17)
```
definef arg1 name arg2; body; end name;
```

Aside from the fact that it is written in a different syntactic form, and that the function name is underlined, the above infix function has exactly the same semantic significance as any two argument function name (arg1,arg2).

A programmer-defined prefix or monadic operator will have only a single argument; its definition will have the form

definef name arg; body; end name;

Infix and prefix operators as well as functions written in ordinary parenthetical style, may freely be used as parts of expressions. Monadic operators always have a precedence superior to that of any infix operator, except that built-in operators computing boolean from non-boolean quantities have higher precedence. The precedence of a given occurrence of a monadic operator may be varied by the use of the "$" convention previously described.

Subroutines of 1 and 2 arguments may also be written in infix and prefix form respectively, having in this case definitions of the form

define arg1 name arg2; body; end name;

and

define name arg; body; end name;

respectively. Subroutines defined in one of these forms should be called in the corresponding form.

We allow subroutines and functions to be called in a variety of ways generalizing the conventional (16). First, as to functions, the call

$$f\ (a_1,\ldots,a_n)$$

will return the value

$$\{f(a_1,\ldots,a_n)\} \quad .$$

This convention is consistent with what we would have if f was a set of ordered n+1-tuples. Of course, if f is a function, the right number of arguments must be supplied to it.

The call

$$f\ [a_1,\ldots,a_n]$$

will return the set

$$\{f(x_1,\ldots,x_n),x_1\varepsilon a_1,x_2\varepsilon a_2,\ldots,x_n\varepsilon a_n\} \quad .$$

Again, this is consistent with our usage in case f is a set of ordered n+1-tuples. Still more generally, a call of the form

$$f(a_1,\ldots,[a_j],a_{j+1},\ldots,[a_k,a_{k+1},\ldots,a_\ell],\ldots)$$

will return the value

$$\{f(a_1,\ldots,x_j,a_{j+1}\ldots,x_k,x_{k+1},\ldots,x_\ell,\ldots),\ x_j\varepsilon a_j,x_k\varepsilon a_k,x_{k+1}\varepsilon a_{k+1},\ldots\}$$

We allow similar forms for subroutine calls. If f is a subroutine, then the call

$$f\ [a_1,\ldots,a_n];$$

is equivalent to the iteration

$$(\forall x_1 \in a_1, x_2 \in a_2, \ldots, x_n \in a_n)\ f(x_1,\ldots,x_n);$$

the call

$$f(a_1,\ldots,[a_j],a_{j+1},\ldots,[a_k,a_{k+1},\ldots,a_\ell]\ldots);$$

is equivalent to the iteration

$$(\forall x_j \in a_j, x_k \in a_k, x_{k+1} \in a_{k+1}, \ldots,)\ f(a_1,\ldots,x_j,a_{j+1},\ldots,x_k,x_{k+1},\ldots,x_\ell\ldots);$$

For monadic operators we can write in very similar fashion and with similar meaning;

$$\underline{op}(a)\quad ;$$

For binary infix operators we can write

$$[a]\ \underline{op}\ b,\quad a\ \underline{op}\ [b],\quad [a]\ \underline{op}\ [b],\ \text{etc.}$$

Subroutines and functions are legitimate atom-types within SETL; in particular, they may validly be elements of sets, components of ordered pairs, arguments of other subroutines, etc. This fact allows various powerful programming devices to be used; we may, for example, tag a set with certain functions of access and combination which are associated

with it. etc.

Aside from the operation of application that naturally belongs to subroutines and functions, the only built-in operations which apply to atoms of these types are the boolean tests eq and ne of equality and inequality.

Each statement in a block of statements occurring in the form

(18) initially block;

will be executed the first time the subroutine containing (18) is entered, but not subsequently. If any such statements occur within a subroutine (or function), they must preceed all other statements of the subroutine (with the exception of declaratory *external* statements; see below). This gives us a general method for the initialization of variables. Within such an *initialization block*, a statement having the degenerate form

```
vname=;
```

is taken as an abbreviation for

```
vname ='vname';
```

and thus initializes the variable vname to a character string equal to the external representationof the variable's name. We also allow the more general form

```
<vname1,vname2,...,vnamek>=   ;
```

this initializes each of the variables whose name occurs within the pointed brackets, each to a character string equal to the external representation of its own name.

A few extra words defining the semantics of initialization blocks (18) somewhat more precisely are in order; especially in case (18) occurs within a recursive subroutine confusion might otherwise arise. We take the initialization (18) to be precisely equivalent to the statement

(19) if flag eq 0 then flag=1; block;

where *flag* is a generated variable uniquely associated with the initialization statement (18). The variable *flag* occurring in (19) is also taken to be external to the subroutine A in which (18) occurs i.e., this variable is not stacked if A is entered recursively.

Thus the statements of *block* will be executed only when A is invoked for the first time, even if A is entered recursively. On the other hand, perfectly arbitrary statements, and even recursive subroutine calls, may be contained in *block*.

In addition to the effects of *explicit initializations* of the form (18), the values of variables will be initialized in certain circumstances for reasons implicitly connected with the usage of these variables. Such *implicit initializations* will be made before any *explicit initializations* of the form (18) are made. The circumstances leading to implicit initializations are as follows:

i. If a name *labname* is used as a label within a SETL

routine, then the variable of the same name will be initialized to have a value equal to the unique label atom corresponding to the label *labname*. This initialization is useful in that it allows us to write such expressions as

```
go to{<const1,lab1>,<const2,lab2>,<const3,lab3>}(expr);
```

and thus to make use of "calculated go-to's" in flexible and convenient form.

ii. A procedure definition A may be imbedded within the text of another procedure B, in which case the procedure A is said to be *internal* to B. If A is internal to B, but not internal to any C which is itself internal to B, then A is said to be *immediately internal* to B. A procedure whose definition is not imbedded in the text of any other SETL procedure is said to be an *external* procedure.

A name A is said to be *known as a procedure name* within a subroutine B if either

a) A is identical with the name of some external procedure; or
b) A is identical with the name of a procedure immediately internal to B; or
c) A is identical with the name of a procedure to which B is immediately internal; or
d) A is identical with the name of B; or
e) A is identical with the name of a procedure C, while C and B are both immediately internal to the same subroutine D; or
f) The name A has been identified by the use of an *external* statement (for which see below) with a name occurring in some other subroutine

C and known as a procedure name within C.

Cases a), b), c), e) are all subject to the proviso that the name A has not been used as a label within B, and that A has not been identified within B with any external name. Concerning d), we note that the name of a subroutine may not be used in the subroutine as a label. Concerning f), we note that the name of a procedure may not be identified in the procedure with any external name; and that a name used within a procedure as a label may not be identified in that procedure with any external name.

In each of the cases a)-f), a name A known as a procedure name will refer to some unique procedure P; when A is known as a procedure, its value is initially set equal to the unique subroutine or function atom corresponding to the procedure P. [Remember that in SETL subroutines and functions are atoms and are often treated in the same way as atoms of any other kind. In particular, a variable whose value is initially a subroutine or function may have its value changed by an assignment statement; for example, if subs is a sequence of single-argument subroutines, we may legally write

```
(1≤ Vn≤ #subs) sub:=subs(n); sub(x);;
```

thereby applying every subroutine in the sequence to the variable x.]

iii. Except in the situations described by i) and ii) above, the value of a variable will initially be undefined.

We now turn to describe the (rather conventional) name-

scoping rules used in SETL, and the *external* statement which may be used to vary the scope of names. A SETL variable or label name is ordinarily local to the (main program or) defined subroutine or function body which contains it. That is, identical names occurring within distinct subroutines A and B normally refer to different variables. The various forms of the *external* statement serve to identify name-references across procedures. The basic form of this statement is

(20) suba external name1, name2,..., namek,

where *external* is a keyword, where *name1,...,namek* are variable names, and where *suba* is a valid SETL name known as a procedure name within the subroutine (or function A) in which (20) occurs; suba is known as the *procedure reference* of the external statement. The statement (20) defines *name1,...,namek* as having references which are the same as those which identical names occurring in the subroutine *suba* would have.

Occasionally one will want to be able to access a variable *namea* defined in a subroutine *suba* from within a subroutine B, but will want the variable to be known under a different name within the subroutine B. This might, for example, be the case if B had occasion to access two different variables, originally occurring in two different subroutines, both of which happened to have the same name. For this purpose the variant form

(20) suba external (namea, nameb),...

is provided; this declaration, occuring within the subroutine

B, identifies the variable namea occuring in suba with the variable nameb occuring in B.

If an *external* statement identifies a name named as a procedure name, then this name may in turn be used as a procedure name in external statements. Thus, for example, we might have

```
define suba(x);
define subsub(y);...; z=x+1;...;end subsub; end suba;
define subb(u,i);suba external subsub;
                 subsub external x;...;end subb;
```

In this case, the first of the two *external* statements occuring in *subb* would make the subroutine *subsub*, which is internal to *suba*, known as a procedure name within *subb*; and the second *external* statement in *subb* then identifies x with a variable of the same name occuring within *subsub*.

The "main program" within which all SETL subroutines occur may be considered to be a subroutine with an "empty" name, the so-called *main routine*. The statement

```
return;
```

occuring in this main subroutine terminates program execution. An *external* statement with a missing procedure reference is taken to refer implicitly to the main routine. That is,

(21) external name1,...,namek

identifies each of name1,...,namek with a variable of the same form occuring in the main routine of the program

containing the subroutine within which (21) occurs. *External* statements referring to the main routine but having the generalized form (20) are also provided.

Note once more that names used as subroutine arguments or as labels may not be declared external. A given name may only be declared as external once within a given subroutine. All the *external* statements occuring within a given procedure must be grouped at the beginning of the procedure, and must precede any other statements in the procedure.

SETL provides a variety of macro forms, ranging from a very simple type of macro allowing program text to be written in a more readable statement order than would otherwise be possible, to rather highly structured syntax macros providing SETL with powerful language extension facilities. At this point in our exposition, we shall only describe the simplest SETL macro feature. This feature allows us to define a named macro-block of code by writing

(22) block name (arg1,...,argn); body; end name;

Here *name*,*arg1*,...,*argn* are valid SETL names, and *body* may be any legal sequence of statements; *arg1*,...,*argn* will normally normally occur within *body* but *name* should not. A macro-block defined in the form (22) can be invoked by writing

(23) do name(tok1,...,tokn);

Where *tok1*,...,*tokn* are valid SETL tokens. These tokens are substituted for *arg1*,...,*argn* in the *body* of (22), and the resulting text replaces the statement (23) at its point

of occurrence.

As indicated, this simple macro-form is provided so that appropriate subsections of SETL programs can be arranged in whatever order most enhances readibility. For example, one might wish to write an if-statement as

```
if a gt b then do abigger (n); else if a lt b then do aless(n);
                    else do equal(n);;
```

especially if the code blocks *abigger* and *aless* are at all lengthy. Note in this connection that the clarity of a code containing macro-blocks can be greatly enhanced by the redundant use of macro parameters. Any parameter forming a significant part of the logical context within which a macro block (22) will be called should appear explicitly in the macro definition even though the actual manner in which the macro is to be used might make this superfluous.

m. Special Conventions Concerning the Atom Ω.

We relate the special undefined atom Ω to the various SETL operations in somewhat special ways. Ω is a blank atom, but one having slightly unusual properties. We do not allow Ω to be a member of any set, so that any attempt to form such a combination as

$$\{\Omega\},\{\Omega,a\} \qquad \langle\Omega,a\rangle \qquad \langle a,\Omega\rangle$$

will lead to an error return. An attempt to form any of

pow (Ω), Ω(x), f(Ω), t(Ω) , f[Ω], Ω , or #Ω

will also lead to an error return. The atom Ω is allowed in the combinations x eq Ω and x ne Ω, but of course not in x gt Ω or in any other arithmetic, string, or boolean operation. (Note that any attempt to execute an operation with arguments of inappropriate type will also lead to an error return.) An occurrence of Ω either in a quantifier of the form

∃ ∀x∈Ω, x∈Ω,Ω≤ ∀k≤ x, etc.

in a range restriction

x∈Ω,Ω≤ k≤ x, etc.,

or in any iteration control of the form

(while Ω)

will lead to an error return. The value Ω will be returned as the value of

<---><a,b>, etc.

if b is not a pair, and as the value of f(a) if a is not in the domain of a set f of ordered pairs. We also have hd Ω = tl Ω = Ω .

These conventions have very useful effect in locating bugs in SETL programs, as they ensure that many situations in which the actual form of data differs from its assumed form will lead rapidly to the occurrence of Ω as a value and very shortly thereafter to an error return.

The assignments

a=Ω; f(a)=Ω; and f(a,b)=Ω; etc.

are legal. The second of these has exactly the same effect as execution of the statement

f = f lesf a ;

n. Comments; Print and Read Statements

Comments are enclosed fore-and-aft by use of the composite marks /* (prefixed) and */ (affixed). Thus, for example

/* this is an example of a setl comment. */.

Input/output conventions sufficiently substantial to allow SETL to make use of externally stored files will be described later. Here, however, we describe two SETL statements giving a rudimentary input/output facility for use in connection with a standard input and a standard print file. These have the form

(24) s print $expr_1$, $expr_2$, ..., $expr_n$

and

(25) s read $name_1$, $name_2$, ..., $name_n$

respectively, where s is either an expression having as value an ordered pair <st,n> consisting of a character string and an integer, or is one of the special systems names input or output.

The form in which a set will be printed is determined by the following recursive conventions. An integer will appear in decimal form, possibly preceded by a minus sign

a character string will appear enclosed within quote marks in its normal external form, quote marks themselves being represented by double quotes. Bit strings will appear either in "binary" forms such as 01100101...01B, or in a combined "binary-octal" form, in which a binary prefix preceeds the letter B and an octal suffix follows it, the total bit-string being the concatenation of these two separately represented parts. Note for example that the strings 10111000b and 10b70 are identical; and that either form may be used in a SETL program to represent a bit-string constant.

If a1,...,an are the elements of a set s, and 1,...,r2 are the printed representatives of a1,...,an respectively, then the set

{s1,...,an}

will generally appear printed as the character string

{r1,...,rn}

a similar convention applying to n-tuples. On the other hand, each set to be printed will be tested to see if it is of the special form

{<1,a1>,<2,a2>,...,<n,an>},

i.e., is a set s all of whose elements are ordered pairs, and such that each integer in the range from 1 to an upper limit n=#s occurs precisely once as the first component of an element of s. The printed representation of a set having this special form will be

{r1,r2,...,rn},

where r1,...,rn are the printed representations of a1,...,an respectively. These simple recursive rules will be used as long as the length of the character strings it produces does not exceed two printed lines, and the level of parenthesis nesting needed within these character strings does not exceed 4. When these limits are exceeded, sub-sets of a composite structure which is to be printed will be assigned abbreviating designators consisting of an integer followed by a decimal point, and the significance of such abbreviations will be indicated on separate printed lines, indentation being used appropriately to improve the readability of the resulting text. Thus, for example, a set that might have been printed as

(26) {{{{5,10,15,<20,{21,22,23,{{24,<25,8>},9}}>,3}}}}

will actually appear as

(27) {{{{5,10,15,1.,3}}}}
1. <20,{21,22,23,{{24,2.},9}>
2. <25,8> .

The SETL read statement (25) will accept, from the standard input string *input*, sets represented in this way, and convert them into the abstract structures which they represent, assigning the set thereby obtained as the value of the variable name occuring in (25). The number of characters of the *input* string digested during such a read operation is determined by the following rule: blank characters, up to a first non-blank character, which must be either{,<, or [will be ignored. All the characters of a balanced-parenthesis

structure of some such form as (23) or (24) will then be digested, blanks (except quoted blanks) being ignored, and single characters as (but not pairs of successive characters as) being ignored also. Once a balanced structure of a form like (23) or (24) has been read, subsequent blanks, up to and including a first as character, or up to but not including any other non-blank character, will be skipped, and the *read* operation then terminated.

9. Dynamic compilation.

SETL programs will occasionally include or generate program text; by allowing such text to be compiled during program execution, we provide SETL yet another quite powerful feature. This feature is provided by a prefix operator

(28) compile x;

the value of whose argument x is a character string. This character string must be a valid subroutine or function definition, i.e., must have either the form

(29) 'define sub...; text; end sub;' ,

if a subroutine is to be compiled, or the form

(30) 'definef fin...; text; end fin;'

if a function is to be compiled. Here *sub* and *fin* must be valid names. Strings (29) and (30) to be compiled may not be infix- or prefix-form operator definitions.

The value of the expression (28) is an atom either of subroutine (in case (29)) or of function type (in case (30)). This atom may then be assigned as the value of a SETL variable, and called in the normal fashion. Thus, for example, we might write

```
symdif-compile 'definef sd(a,b);
  return {x:a | x in b} with {x:b | x in a};
  end sd;'; x=symdif({a,b,c},{b,d,e});
```

This would give x the value {a,c,d,e}.

The *text* appearing either in (29) or in (30) may contain imbedded subroutine and function definitions, *external* statements, and *initially* statements.

The name-scoping rules which apply to dynamically compiled subroutines may be stated as follows. Suppose that the statement (28) appears in the subroutine A of a complete SETL program. Then, when (28) is performed, name scopes will be assigned within the resulting subroutine using precisely the rules that would have applied if the text x had originally appeared as an imbedded subroutine definition within A.

p. General extraction and replacement operators. Generalized multiassignments

In the present section, we complete our account of the basic parts of the SETL language by describing extraction operators related to those specified in section a) above, but considerably more general and intended to be of convenient use in connection with nested structures built up out of ordered n-tuples. Generalized forms related to the extraction operators which will be described will also be used in writing multiple assignment statements. We shall also describe a family of structured replacement operators, which allow the replacement of particular components of ordered n-tuples by specified values; those replacement operators will be useful in the same situations as the generalized extraction operators with which we shall be concerned. Note that the general extraction operators to be described in the present section generalize the simple and the nested selection operators described in section a) above, and the assignment bracket and assignment operators described in section j) above. The following principal new features are provided by the generalized forms described in this section.

i. Specification of component by numerical index, e.g.

<* g 13> x

is the 13-th component of the n-tuple x;

<* g n> x

is the n-th component of x.

ii. Specification of components of components by addressing sequences, e.g.

<* $\underline{s}$ <2,3,5>> x

extracts the 5-th component of the 3-rd component of the 2-nd component of x.

iii. Nesting to arbitrary depth, and the use of names and indexed names to get 'assignment' effects.

The detailed specification of the semantics of these rather general extraction operators involves a mass of fairly complex detail, and makes a careful and precise description necessary. We begin by explaining the notion of *structural address*. In an n-tuple of the basic form

<a,b,c,...,e>

the structural address of any component is its position in the n-tuple. Thus a has the structural address 1, c the structural address 3, e the structural address 5. If the j-th component of an n-tuple x is an m-tuple y, then the k-th component of y has the structural address j,k in x, and we allow this construction to be carried to any depth of nesting. Thus, for example, in

(1) <<a,b>,<<< c,d>,e>,f>,g>

b has the structural address 1,2; d the structural address 2,1,1,2; e the structural address 2,1,2; f the structural address 2,2; and g the structural address 3. We may also take the structural address 2,1 to refer to <<c,d>,e>, and the structural address 2,1,1 to refer to <c,d>,etc. Note that since such identities as

<a,b,c,d> = <a,b,<c,d>> ,

every n-tuple may also be regarded as an m-tuple for every m < n; from this arises a certain irritating tendency toward ambiguity in the interpretation of structural addresses, a tendency which we must be prepared to combat. We remark that the complex collections of n-tuples which form the operands for highly

structured extraction operators will in general be "tables". In such tables, the order in which particular entries are given is a matter of convention. A structural address serves to define the location of an entry in such a table; the value of this entry is then assigned to the appropriate target in the extraction operator.

At any rate, each finite sequence of positive integers is a structural address.

Each structural address

(2) $$sa = n_1, n_2, \ldots, n_k$$

consisting exclusively of positive integers defines two operations: a *short selection* O_{sa} and a *full selection* $\bar{O}_{sa}$. These operators are defined by

(2') $$O_{sa} = t_{n_k}\, t_{n_{k-1}} \cdots t_{n_2}\, t_{n_1}$$

and

(2'') $$\bar{O}_{sa} = \bar{t}_{n_k}\, t_{n_{k-1}} \cdots t_{n_2}\, t_{n_1},$$

where t_m and $\bar{t}_m$ are defined by

(2''') $$t_1 = \underline{hd},\ \bar{t}_1 = \text{identity};$$
$$t_m = \underline{hd}(\underline{tl})^{m-1},\ \bar{t}_m = (\underline{tl})^{m-1} \qquad \text{for } m > 1.$$

Note for example that

$$O_{3,2} = t_2 t_3 = (\underline{hd}\ \underline{tl})\ (\underline{hd}(\underline{tl})^2)$$
$$= \underline{hd}\ \underline{tl}\ \underline{hd}\ \underline{tl}\ \underline{tl},$$

and similarly $\bar{O}_{3,2} = \underline{tl}\ \underline{hd}\ \underline{tl}\ \underline{tl}$, so that

$$O_{3,2}\langle a,b,\langle c,d,e\rangle,f\rangle$$

is d whereas

$$\bar{O}_{3,2}\langle a,b,\langle c,d,e\rangle,f\rangle$$

is $\langle d,e\rangle$. The selection operations O_{sa} and $\bar{O}_{sa}$ will be used below to define the semantics of our generalized extraction operators.

Next we describe the way in which generalized extraction operators are built up. Each such operator has the form

(3) $\langle part_1, part_2, \ldots, part_n \rangle$,

where each *part* has one of the forms

(4) name, name, z expn, name zt expn, *, * z expn, * zt expn, -, n-, or

(5) exop or exop expn,

where *exop* is itself an extraction operator. The symbol z is read "is"; the symbol zt is read "is tail". In (4), *name* may either be a simple name, or may be an *indexed name* of one of the forms

name(exp), name(exp),..., name(exp_1,exp_2), etc.

Again in (4), each expn has an m-tuple of positive integers as a value, and n has a positive integer as value. The asterisk, in one of its usages (4), will be called the "special name"; this name is allowed to appear only once in an extraction operator (3). Note that the asterisk in (4) is used in much the same way as a name. The value assigned to this name is the value of the expression $\langle part_1, \ldots, part_n \rangle$"* .

An extraction operator (3) associates a structural address with each part and each simple or indexed name occurring within it, according to the following rules:

(a) if a part of (3) has the form

name z expn, name zt expn, * z expn, or * zt expn,

the structural address which (3) associates with *name* (or with the special name *) is the sequence of components of the m-tuple obtained by evaluating *expn*.

(b) if a part of (3) has the form

exop z expn,

then the structural address $n_1, \ldots, n_k$ which (3) associates with *exop* is determined as in the preceding case (a). Then, recursively, if *exop* would associate the structural address

$\bar{n}_1,\ldots,\bar{n}_k$ with a subpart occurring within it, then (3) associates the concatenated sequence $n_1,\ldots,n_k,\bar{n}_1,\ldots,\bar{n}_k$ with the same subpart.

(c) if the k-th part of (3) has the form

name . or * ,

the structural address which (3) associates with *name* (or with *) is the sum of the *place values* of all preceding parts of (3). We call this the *place address* which (3) associates with *name* (or with *). Place values are defined as follows: a part of the form

n-

has place value n; a part of any other form has the place value 1. (Note that n is allowed to be any expression with a nonnegative integer value).

A part of (3) having the form

name

is taken to have the same significance as a part of (3) having the form

name $\underline{\text{z}}$ n ,

unless *name* is the last part of (3), in which case it is taken to have the same significance as

name $\underline{\text{zt}}$ n;

here n is the place address of *name* in (3). Similarly, a part of (3) having the form

*

is taken to have the same significance as

* $\underline{\text{z}}$ n,

unless * is the last part of (3), in which case it has the significance

* $\underline{\text{zt}}$ n .

(d) if the k-th part of (3) has the form

exop ,

the structural address n which (3) associates with *exop* is determined as in the preceding case (c). Then, recursively, if exop would associate the structural address $\bar{n}_1,\ldots,\bar{n}_k$ with

a subpart occurring within it, then (3) associates the concatenated sequence $\bar{n}_1,\ldots,\bar{n}_k$, n with this same subpart.

Note also that the extraction operator

<<a,b>,c,*>

might also be written as

<<a $\underline{\geq}$ 1, b $\underline{\text{gt}}$ 2>, c $\underline{\geq}$ 2, * $\underline{\text{gt}}$ 3>

and that the operator <x,y> in a multiple assignment statement might also be written <y $\underline{\text{gt}}$ 2, x $\underline{\geq}$ 1>.

(e) A part of either of the forms

- or n-

has no associated structural address, but is included either for its effect on the place values of the other parts of (3), or is included to prevent some other part of (3) from being the last.

If (3) associates the structural address $n_1,\ldots,n_k$ with a certain *name*, and if *name* occurs in (3) either explicitly or implicitly in the form *name* $\geq$ *expn*, then when the assignment operator (3) is applied to a quantity x the assignment

(6) $$\text{name} \leftarrow \mathcal{O}_{n_1,\ldots,n_k}\ x$$

is made. On the other hand, the assignment

(6') $$\text{name} \leftarrow \bar{\mathcal{O}}_{n_1,\ldots,n_k}\ x$$

is made if *name* occurs in (3) either explicitly or implicitly in the form *name* $\underline{\text{gt}}$ *expn*. Note once more that $\mathcal{O}_k\langle c_1,\ldots,c_k,\ldots,c_m\rangle$ is c_k, while $\bar{\mathcal{O}}_k\langle c_1,\ldots,c_k,\ldots,c_m\rangle$ is $\langle c_k,\ldots,c_m\rangle$, the "tail" from c_k on.

If *name* in (6) or (6') is indexed, then all indices in (6) or (6') are evaluated before any assignments (6) or (6') are made. Note in particular that if c = 1 before the statement

<c,b(c)> = <2,1>;

is performed, the result is equivalent to that of the statements

b(1) = 1; c = 2;

In any case, assignments (6) or (6') are made in sequence in which the corresponding names occur in (3), in left-to-right order. Note also that if the value of *expn* in (6) or (6') is such that $\sigma_{n_1,\ldots,n_k}$ expn or $(\bar{\sigma}_{n_1,\ldots,n_k}$ expn) is undefined, the assignment name ← Ω is made.

An extraction operator (3) is applied to *expn* in the following syntactic form. If (3) contains no occurrence of the "special name" *, we write

(7) $\quad$ <part$_1$,...,part$_n$> = expn;

i.e., use a *generalized multi-assignment* form. If (3) contains a (necessarily unique) occurrence of the "special name" *, we use

(8) $\quad$ ... ; <part$_1$,...,part$_n$>expn ...

within some other expression. When this *generalized extraction operator* is used, various assignments (6) and (6') will be made; assuming that (3) associates the structural address $n_1,\ldots,n_k$ with the special name, the value of the expression (8) will be either $\sigma_{n_1,\ldots,n_k}$ expn or $\bar{\sigma}_{n_1,\ldots,n_k}$ expn, according to the rule explained above in connection with (6) and (6'); i.e. the value of the expression (8) is the value that would be assigned to the special variable *, viewed as a variable name.

As an example of the use of our extraction operators, note that the following subroutine produces the sequence of components of an n-tuple

```
defineF seq tuple; s←nil; j←1;
    (while <* ≥ j,-> tuple ne Ω doing j←j+1)
        <s(j) ≥ j, -> ← tuple;; return s; end seq;
```

An equivalent program, written rather differently, is given in section II C below.

Note that when the extraction operator is used in the form (7) we regard it as a monadic operator, subject to the usual rules concerning operator scope. In particular, the precedence of an

extraction operator (8) may be raised by affixing the sign $ to it, and lowered by prefixing $ to it.

Next we turn to describe replacement operators. These operators have the form

(9) $\langle part_1,\ldots,part_n \rangle$,

and are used as monadic operators within expressions, i.e., in the basic combination

(10) $\langle part1,\ldots,part_n \rangle$ expn .

These operators serve in various ways for the replacement of components of an n-tuple. In particular, the following possibilities are provided.

i. Replacement of components specified by position, e.g.

<-,a r,-,b,-> <c,d,e,f,g> is <c,a,e,b,g> .

ii. Replacement of all components past a given one by a specified expression, e.g.

<-,x+y rt 3> <c,d,e,f,g> is <c,d,x+y> .

as this example indicates, the point of replacement may be specified by an expression; and the replacement quantity specified by an expression also.

iii. Specification of components of components by addressing sequences, e.g.

<a rt <2,2>> <c,<d,e,f>,g> is <c,<d,a>,g> .

iv. Nesting to arbitrary depth. Thus, for example,

<<a r 2,-,b>, e> <<d,e,f,g>,h,i> is <<d,a,b,g>,c> .

Each part of a replacement operator has one of the forms

(11) exp, exp r, exp r expn, exp rt, exp rt expn, -, n-, or

(12) ropn , or ropn expn

where *ropn* is itself a replacement operator. In (11) and (12), *expn* is any expression whose value is an n-tuple of positive integers. We require in any case that the symbol r occur at least once in each replacement operator (8).

Conventions almost precisely like those given above for the case of the extraction operator associate the following constructs to each replacement operator:

i. A collection of structural addresses belonging to the replacement operator (9);

ii. A rule which associates a structural address with each exp occurring in a part or subpart of the replacement operator (9). We require that these structural addresses contain no zeros, i.e. that they be sequences $n_1,\ldots,n_k$ of positive integers.

If a structural address $n_1,\ldots,n_k$ is associated with an expression exp by the replacement operator (9), and if exp appears in (9) or one of its subparts either implicitly or explicitly in the form exp zf expB , then we require that then no structural address $n_1,\ldots,n_{k-1},n_k'$ with $n_k' \geq n_k$ be associated with any other expression exp' by (9).

Suppose now that the collection S of all the structural addresses covered by the preceding remark iii is arranged in order of decreasing length, and that we let $R_{n_1,\ldots,n_k}$ be the value of the replacement expression exp associated with $n_1,\ldots,n_k$ as in ii. Then the value of (9) may be defined recursively, as follows.

(a) First take all the structural addresses $n_1,\ldots,n_k$ in S and of maximal length k. Group all those structural addresses of maximal length which begin with a given initial sequence $n_1,\ldots,n_{k-1}$ together into a set J, and write

$$C_j = R_{n_1,\ldots,n_{k-1}},$$

whenever $sa = n_1,\ldots,n_{k-1},j$ belongs to J. If on the other hand sa is a structural address of length k beginning with the sequence $n_1,\ldots,n_{k-1}$ which does not belong to J, but some element of J exceeds sa, put

$$C_j = O_{n_1,\ldots,n_{k-1},j}^{\text{exp}}$$

cf. (10) and (2'), (2") above. Let $n_1,\ldots,n_k$ be the largest element of J, in lexicographic order. If this structural index

is associated with exp by the replacement operator (9), and if exp occurs within (9) either explicitly or implicitly in the form exp λt ... , then put

$$P_{n_1,\ldots,n_{k-1}} = \langle C_1,\ldots,C_{n_k}\rangle .$$

If, on the other hand, exp occurs in (9) either explicitly or implicitly in the form exp λ ... , put

$$P_{n_1,\ldots,n_{k-1}} = \langle C_1,\ldots,C_{n_k},\bar{O}_{n_1,\ldots,n_{k-1},n_k+1}\ \text{expn}\rangle$$

if $\bar{O}_{n_1,\ldots,n_{k-1},n_k+1}$ expn is not Ω, otherwise put

$$P_{n_1,\ldots,n_{k-1}} = \langle C_1,\ldots,C_{n_k}\rangle .$$

(b) Next, supposing that $P_{n_1,\ldots,n_k}$ has been defined whenever $k \geq q$, we are prepared to treat structural indices of length q. Group all the structural address of length q which begin with a given initial sequence $n_1,\ldots,n_{q-1}$ into a set J_0, and let J be the set of elements of J_0 which form the initial part of some sequence S. Then, for each structural address $sa = n_1,\ldots,n_{q-1},j$ in J, the quantity $P_{n_1,\ldots,n_{q-1},j}$ will already have been defined. Now put

$$C_j = P_{n_1,\ldots,n_{q-1},j}$$

if sa belongs to J but not to S, but put

$$C_j = R_{n_1,\ldots,n_{q-1},j}$$

if sa belongs to S. If on the other hand sa belongs to J_0 but not to J, while some element of J exceeds sa, put

$$C_j = O_{n_1,\ldots,n_{q-1},j}\ \text{expn}$$

cf. (10) and (2'), (2") above. Let $n_1,\ldots,n_k$ be the largest element of J. If this structural index is associated with exp by the replacement operator (9), and if exp occurs within (9) either explicitly or implicitly in the form exp λt ... , then put

$$P_{n_1,\ldots,n_{k-1}} = \langle C_1,\ldots,C_{n_k}\rangle .$$

If, on the other hand, *exp* occurs in (9) either explicitly or implicitly in the form *exp x ... ,* put

$$P_{n_1,\ldots,n_{k-1}} = \langle C_1,\ldots,C_{n_k},\bar{\delta}_{n_1,\ldots,n_{k-1},n_k+1}\ \text{expn}\rangle$$

if $\bar{\delta}_{n_1,\ldots,n_{k-1},n_k+1}$ expn is not Ω, otherwise put

$$P_{n_1,\ldots,n_{k-1}} = \langle C_1,\ldots,C_{n_k}\rangle .$$

(c) The above rules define $P_{n_1,\ldots,n_k}$ for all sequences $n_1,\ldots,n_k$, and, in particular, associate a well-defined quantity P with the null sequence. This quantity P is, by definition, the value of the expression (10).

A few additional examples will help clarify the effect of the rules stated above.

For example, if *data* is <<c,d,e>,<f,g,h>,<i,j>,k>, then

<a <u>r</u><1,2>, b <u>rt</u> 3> data

has the value

<<c,a,e>,<f,g,h>,b> ,

and

<<-,a,->1,b <u>rt</u> 3> data

has the same value, while

<<-,a>1, b <u>r</u> 3> data

and

<a <u>rt</u><1,2>, b <u>r</u> 3> data

both have the value

<<c,a>,<f,g,h>,b,k> .

If *tup* is an n-tuple, the following will define an n+1 tuple whose first n coordinates are the same as those of *tup*, and whose n+1-st coordinate is x:

newtup = <x <u>r</u> n+1> tup;

this may also be written as

newtup = <x <u>rt</u> n+1> tup;

Again, to set newtuple to $\langle c_1,\ldots,c_{j-1},x,c_j,\ldots,c_n\rangle$, where *tup* is $\langle c_1,\ldots,c_n\rangle$, i.e., to insert an additional component at the n-th position, execute

```
newtuple = <x ↑ j, <* ↓ j>tup ↓(j+1)>tup;
```

which of course involves both replacement and extraction operators. Alternatively, execute

```
newtuple = <(j-1)-, x ↑, <(j-1)-,*>tup>tup;
```

This is of course the same operator.

IIB. Recapitulation of the basic parts of the SETL language.

In the present section, we recapitulate, in capsule form, the principal basic features of the SETL language. While this merely repeats information given in considerably more detail in the preceding section, it may be hoped that such a *precis* may serve as a useful brief reference for the reader.

Basic objects: *Sets* and *atoms*; sets may have atoms or sets as members. Atoms may be

Integer. examples: 0, 2, -3

Boolean strings. examples: 1b, 0b, 77o, 00b777

Character strings examples: 'aeiou', 'spaces-

Label. (of statement) examples: label:, [label:]

Blank. (created by function newat)

Note: Special undefined blank atom is Ω.

Subroutine. Function.

Basic operations for atoms:

Integers: arithmetic: +, -, *, /, // (remainder)

comparison: eq, ne, lt, gt, ge, le

other: max, min, abs

Examples: 5//2 is 1; 3 max -1 is 3; abs -2 is 2.

Booleans: logical: and (or &), or, exor, implies (or imp), not (or !)

logical constants t (or true, or 1b);
f (or false, or 0b).

Character strings: conversion: dec, oct

Examples: dec '12' is 12; oct '12' is 10.

Strings (character or boolean):

+ (catenation), * (repetition), first, last, elt (extraction)

len (size), nul, nulc (empty strings).

Examples: 'a' + 'b' is 'ab'; 2 * 1b4 is 1100110b;

2 * 'ab' is 'abab', 2 first 'abc' is 'ab',

2 last 'abc' is 'bc', 2 elt 'abc' is 'b',

len 'abc' is 3, len nul is 0.

General: Any two atoms may be compared using eq or ne;

atom a tests if a is an atom.

Basic operations for sets.

ε (membership test); nl (empty set); ∃ (arbitrary element),

(number of elements); eq, ne (equality tests);

incs (inclusion test); with, less (addition and deletion

of element); lesf (ordered pair deletion).

pow(a) (set of all subsets of a);

npow(k,a) (set of all subsets of a having exactly k elements).

Examples: a ε {a,b} is t, a ε nl is f, ∃ nl is Ω,

∃ {a,b} is either a or b, # {a,b} is 2, # nl is 0,

{b} with a is {a,b}, {a,b} less a is {b},

{a,b} less c is {a,b}, {a,b} incs {a} is t.

pow({a,b}) is {nl, {a}, {b}, {a,b}}.

npow(2,{a,b,c}) is {{a,b}, {a,c}, {b,c}}.

Ordered pairs: <a,b> first and second component extractors are hd, tl; n-tuples <a,<b,c,...,d>> = <a,b,c,...,d>

Examples: hd<a,b> is a, tl<a,b> is b,

hd<a,b,c> is a, tl<a,b,c> is <b,c>; (hd p is p if p not pair).

Note that <a,b> is identical with {{a},{a,b}}, so that for example {a} ε <a,b> is t while a ε <a,b> is generally f.

See also: extraction operators, generalized extraction operators, replacement operators, and multi-assignment statements.

Set-definition: by enumeration {a,b,...,c}

Set former:

$\{e(x_1,\ldots,x_n),\ x_1 \in a_1,\ x_2 \in a_2(x_1),\ldots,\ x_n \in a_n(x_1,\ldots,x_{n-1}) \mid C(x_1,\ldots,x_n)\}$.

The *range restrictions* x ε a(y) have the alternate *numerical form*

$$\min(y) \leq x \leq \max(y)$$

when a(y) is an interval of integers.

Optional forms include {x ε a | C(x)}, equivalent to {x, x ε a | C(x)}; and {e(x), x ε a}, equivalent to {e(x), x ε a | t}.

Functional application: (of a set of ordered pairs, or a programmed, value-returning function)

f{a} is {tl p, p ε f | (hd p) eq a}; i.e. is the set of all x such that <a,x> ε f

f(a) is : if # f{a} eq 1 then ? f{a} else Ω, i.e., is the unique element of f{a}, or is undefined atom.

f[a] is {tl p, p ε f | (hd p) ε a}, ie, the image of a under f.

More generally,

f(a,b) is g(b) and f(a,b) is g(b), where g is f(a);

f[a,b] is {tl tl g, g∈f | (hd g)∈a and ((hd tl g)∈b)} .

If f is value-returning function,

f(a,b) = {f(a,b), f[a] = {f(x), x∈a), etc.

Constructions like f[a,[b],c], etc. are also provided.

Compound operator:

{op: x∈s}e(x) is e(x_1) op e(x_2) op... op e(x_n),

where s is{x_1,...,x_n}.

This construction is also provided in the general form

{op: x_1∈e_1,x_2∈e_2(x_1),...,x_n∈e_n(x_1,...,x_{n-1}) | C(x_1...x_n)},

where the range restrictions may also have the alternate numerical form.

Examples: {max: x∈{1,3,2}}(x+1) is 4,

{+: x∈{1,3,2}}(x+1) is 9,

{+: 1≤i≤n}a(i) is SETL form of $\sum_{i=1}^{n} a_i$.

{op: x ∈ nl} e(x) is Ω.

Quantified boolean expressions:

∃x∈a|C(x) ∀x∈a|C(x)

general form is

∃x_1∈a_1,x_2∈a_2(x_1), ∀x_3∈a_3(x_1,x_2),... |C(x_1,...x_n),

where the range restrictions may also have the alternate numerical form.

Search with assignment:

$\exists[x]\in a|C(x)$ has same value as $\exists x\in a|C(x)$, but sets x to first value found such that C(x) eq t. If no such value, x becomes Ω.

Any number of variables attached to initial $\exists$ quantifiers may be placed in square brackets.

Alternate forms

$min\leq\exists[x]\leq max$, $max\geq\exists[x]\geq min$, $max\geq\exists[x]> min$, etc.

of range restrictions may be used to control order of search.

Conditional expressions:

if $bool_1$ then $expn_1$ else if $bool_2$ then $expn_2$... else $expn_n$.

Generalized extraction and replacement operators; generalized multiassignments.

The *extraction operator* has the form

(1) $\langle part_1,\ldots,part_n\rangle$

where each *part* has one of the forms

name, name $\underline{\in}$ expn, name $\underline{\in t}$ expn, *, * $\underline{\in}$ expn, * $\underline{\in t}$ expn, -, n-, or

exop or exop expn, where exop is itself an extraction operator. *Name* may be a simple name or may be an *indexed* name of one of the forms

name (exp), name (exp), name (exp_1,exp_2), etc.

Each exp has an m-tuple of positive integers as a value. Such an operator associates a sequence of integers, called a *structural address*, with each name which occurs within it. Example: in the operator

<<a ≥ 3, b> ≤ <1,2>, *>

the sequence 1,2,3 is associated with a; 1,2,2 with b; and 2 with *. The asterisk * may be used as a name at most once in an extraction operator. The structural address $n_1,\ldots,n_k$ associated with a name (or with the "special name" *) by an extraction operator (1) determines the quantity that will be assigned to the name when (1) is used either in the form

<$part_1,\ldots,part_n$> expr (if * is used once as a name)

or in the form

<$part_1,\ldots,part_n$> =expr? (if * is not used as a name).

Examples:

x = <*,-,iz<2,1>,w><a,<b,c,d>,e,f,g>

results in the assignments

x←a, i←b, w←<f,g>;

x←<*,-,iz<2,1>,w,->><a,<b,c,d>,e,f,g>

results in the assignments x←a, i←b, w←f.

The *replacement operator* has the form (1), where each part has one of the forms

exp r expn, exp r, exp, -, n-, exp rt, exp rt expn

or is itself a replacement operator. At least one occurrence of r is required. Each expn has an n-tuple of positive integers as a value. Such an operator associates a structural address with each exp which occurs within it; the rules for calculating this address are the same as those applying to extraction operators. When a replacement operator is applied to a structure built up in nested fashion out of n-tuples, any element of the structure addressed by a structural address A is replaced by the exp to which A belongs. Each A directs replacement of individual component; each At, an entire "tail".

Examples:

<x,y r 3,->><a,b,<c,d>,e> has the value <x,b,y,e>;

<x,y rt 3><a,b,<c,d>,e> has the value <x,b,y>;

<x,y r<3,1>><a,b,<c,d>,e> has the value <x,b,<y,d>,e> .

Statements: are punctuated with semicolons.

Assignment and multiple assignment statements:

a = expn;

f(exp)= expn; is same as

f =(pcf|(hd p)ne exp}u{<exp,x>,xrexpn);

f(exp) = expn; is same as f{exp}=(expn);

f(a,b) = expn; f{a,b}= expn; etc. also are provided.

<a,b>= expn; is same as a=hd expn; b=tl expn;

<a,b,...,c>= expn;<a,<b,c>,...d>= expn; etc. are also provided.

<f(a),g(b)>=expn; is same as

f(a) = hd expn; g(b)= tl expn;

generalized forms

<f(a), g(b,c),...,h(d)>= expn;

<f(a),<g(b,c),h(d)>,...,k(e)>= expn;

etc. are also provided.

Control statements:

go to label;

if $cond_1$ then $block_1$ else if $cond_2$ then $block_2$...else $block_n$;

if $cond_1$ then $block_1$ else... else if $cond_n$ then $block_n$;

Iteration headers:

(while cond) block;

(while cond doing blocks) block;

$(\forall x_1 \varepsilon a_1, x_2 \varepsilon a_2(x_1), \ldots, x_n \varepsilon a_n(x_1, \ldots, x_{n-1}) | C(x_1, \ldots, x_n))$ block;

in this last form, the range restrictions may have such alternate numerical forms as

min≤ x ≤max, max≥ x≥ min, min≤ x< max, etc.,

which control the iteration order.

Scopes:

The scope of an iteration or of an *else* or *then* block may be indicated either with a semicolon, with parentheses, or in one of the following forms:

end ∀; end while; end else; end if; etc.;
or: end ∀x; end while x; end if x; etc.
or: (∀xεa) til done; block done:...
(while cond) til done; block done:... etc.

Loop control:

quit; quit ∀x; quit while; quit while x;

and

continue; continue ∀x; continue while; continue while x;

Subroutines and functions (are always recursive)

To call subroutine:

sub(param$_1$,....param$_n$);
sub[a]; is equivalent to (∀xεa) sub(x);;

generalized forms

sub(param$_1$,[param$_2$,param$_3$],...,param$_x$)

are also provided.

To define subroutines and functions:

subroutine:

define sub(a,b,c); text; end sub;

return; - used for subroutine return

function:

definef fun(a,b,c); text; end fun;

return val; -used for function return

infix and prefix forms:

define a infsub b; text; end infsub;

definef a infin b; text; end infin;

define prefsub a; text; end prefsub;

definef prefun a; text; end prefun;

Name scopes:

Normally internal to main routine or subroutine, unless declared *external*.

External declarations:

external a,b,c,...; - refers to main routine

suba external a,b,c,...; - refers to subroutine suba

external (a,aa),(b,bb),...; - changes name

suba external (a,aa),(b,bb),...; - changes name a in suba to a, etc.

Macro blocks:

To define a block:

block mac(a,b); text; end mac;

To use:

do mac(c,d);

Initialization:

initially block; (block expected only first time process entered)

In block name=; same as name='name';

Compile function: compile defn has as value subroutine or function defined by character string defn e.g., after executing

y=compile 'definef fromf(x); x= >s; s=s less x; return x; end fromf;');

we may write arb = y set; .

Input-Output:

Unformatted character string:

er is end record character; input, output are standard I/O media; record (s); -- reads till er character from character string sf, at position n, where e is pair<st,n>.

Standard format I/O:

s read $name_1,\ldots,name_n$; reads sets from s=<st,n> at position n as above, or if s is omitted, from input, in standard format

s print $expn_1,\ldots,expn_n$; prints sets on s=<st,n> at position n as above, or if s is omitted on output, in standard format

Ordered sets: $[a_1,\ldots,a_n]$ is i/o representation of $\{<1,a_1>,\ldots,<n,a_n>\}$.

II. C. Examples of the Use of SETL

1. Elementary Examples

We now wish to illustrate the use of SETL, and begin by giving a few elementary definitions of functions to be used later, thereby illustrating, among other things, the definitional facilities of SETL.

The union of two sets is readily defined; before giving this definition it is convenient to define an auxiliary insertion operator, as follows:

```
define a in b; b=b with a; return; end in;
```

The definition of the union of two sets may now be made as follows:

```
define a u b; c=a; (∀ x∈b) x in c; return c; end u;
```

The intersection of two sets may be defined in a number of ways. Perhaps the simplest definition is

```
definef a int b; return {x∈a|x∈b}; end int;
```

A more efficient procedure may be specified as follows:

```
definef a int b; return if #a lt #b then {x∈a|x∈b} else
  {x∈b|x∈a}; end int;
```

In certain of the algorithms to be set forth below, it will be useful to have a function which chooses an arbitrary element from a set, removing the element from the set at the same time. Such a subroutine may be defined as follows:

```
define x from s; x=ɛs; s=s less x; return; end from;
```

2. Sequences, lists, and trees.

A sequence in SETL is a set {<n,xn>} of ordered pairs, the first element of each pair being an integer. Note that sequences and n-tuples are quite different entities.

a. To make an n-tuple of a sequence

```
definef maketup seq; tuple=seq(#seq); (#seq> ∀n>1) tuple
  =<seq(n),tuple>;; return tuple; end maketup;
```

b. To make a sequence of an n-tuple:

```
definef makeseq tuple; seq=Λ; (while pair tuple)
  <seq(#seq+1),tuple> =:tuple;; seq(#seq+1)=:tuple;
  return seq; end makeseq;
```

c. To use a sequence as a pushdown stack, the following conventions are appropriate: Add an element *elt* to the end by executing

```
seq(#seq+1) = elt;
```

Take the top element from the stack by executing

```
top=seq(#seq); seq(#seq)=Ω;
```

A unilateral list may be regarded as a set of items and a function next(item), such that next(item) = Ω for the last item. The basic operations are insertion after a given position and deletion of the next item after a given position. Note that the set of items in the list is the domain of *next*, so that the list is completely determined once *next* is specified. These procedures may be written as follows.

```
define item insafter prev; external next;
 <next(item),next(prev)>=:<next(prev),item>; return;
  end insafter;
define delafter item; external next; nx=next(item);
  if nx eq Ω then return;; next(item)=:next(nx); next(nx)=Ω;
  return; end delafter;
```

A bilateral, circularly linked list may be regarded as a set of items with functions next(item),prev(item) defining the successor and predecessor of a given item; the last item is considered to be the predecessor of the first item, and the first the successor of the last. The first item on the list belongs to a set *firsts*. Note that the three objects *next*, *prev*, and *firsts* together specify the list. The basic operations are insertion of an item after a given position and deletion of a given item. These procedures may be written as follows.

```
define item inabilat prec; external next, prev;
<next(item),next(prec),prev(item),prev(next(prec))>=
   <next(prec),item,prec,item>; return; end inabilat;

define delbilat item; external next, prev;
<next(prev(item)),prev(next(item))>=<next(item),prev(item)>;
if item ε firsts then next(item) in firsts;;
   firsts=firsts less item;
next(item)=Ω; prev(item)=Ω; return; end delbilat;
```

A binary tree is a set of nodes and two descendant functions r and l (right and left descendants); a given top node *ntop* must also be specified. The tree is then entirely defined by these two functions and the specified top node. We take as an example the left-top-right traversal order, and generate the sequence of nodes in the order traversed.

```
seq=nl; traverse ntop;
define traverse top; external seq,r,l; if top eq Ω then return;;
traverse l(top); seq(#seq+1)=top; traverse r(top);
return; end traverse;
```

An ordered tree is a set of nodes with a descendant function desc(node,j) defined for j in some finite (possibly null) range. The ordered and the binary trees stand in an interesting 1-1 relationship. Given an ordered tree, it may be converted to a binary tree by designating its first descendant as its left descendant; and designating desc(n,j+1) as the right descendan of desc(n,j). In SETL:

```
l = {<n,desc(n,1)>, n ε tree | desc(n,1) ne Ω};
r = {<desc(n,j),desc(n,j+1)>, n ε tree, 1 ≤ j ≤ #desc(n)-1};
```

To invert the above transformation, one designates l(n) and the successive right descendants of l(n) in a binary tree as the successive descendants of n. In SETL

```
desc=nl; (∀ netree) k=1; d=l(n); (while d ne Ω doing k=k+1;
 d=r(d);) desc(n,k)=d;; end ∀n;
```

To form an isomorphic copy of a binary tree, the following procedure may be used. Note that a similar procedure will serve to form an isomorphic copy of any structured object:

```
copy=(<n,newat>,nttree); l=lu (<copy(n),copy(l(n))>,nctree);
r=r u (<copy(n),copy(r(n))>, n ε tree);
```

A *threaded tree* is represented by a set *tree* on which two functions r(node) and l(node) are defined, each for all but one node. The values of each of these functions are ordered pairs. We have r(node) = <node',flag>, where *node'* is either the right descendant of *node* or its successor in left-top-right traversal order, depending on whether flag eq t or flag eq f. Similarly, l(node) = <node',flag>, where *node'* is either the left descendant or the traversal-order predecessor of *node*.

To traverse a threaded tree in left-top-right order

```
seq=nl; node=top; dleft: (while(if l(node) ne Ω
  then l(node) else f) node = hd l(node));;
right: seq(#seq+1) = node; if r(node) eq Ω then go to done;;
<node,x>= r(node); go to if x then dleft else right; done; ...
```

To insert an element in a threaded tree, first as the right descendant of a node *nod* :

```
define elt putright nod; external l,r; <r(elt),l(elt),r(nod)>=
<r(nod),<nod,f>,<elt,t>>; return; end putright;
```

To insert an element as the left descendant of a node *nod* :

```
define elt putleft nod; external l,r; <r(elt),l(elt),l(nod)>=
<<nod,f>, l(nod),<elt,t>>; return; end putleft;
```

To delete a node and the entire subtree which this node dominates, we proceed as follows.

```
define delnode node; external l,r;
/* first find the parent node of the given node,
   using either 'ltarg' or 'rtarg' */
n=node; (while l(n) ne Ω and  tl l(n)) n=hd l(n);; ltarg=hd l(n);
n=node; (while r(n) ne Ω and  tl r(n)) n=hd r(n);; rtarg=hd r(n);
/* then change the right or left pointer of the parent,
   as appropriate */
if (if ltarg ne Ω then hd r(ltarg) else Ω)  eq node
   then r(ltarg)=<rtarg,f>; deltrav node;
else if (if rtarg ne Ω then hd l(rtarg) else Ω)  eq node
   then l(rtarg)=<ltarg,f>; deltrav node;
/* if no parent, remove the whole tree */
else <l,r>=<nl,nl>;; return; end delnode;
define deltrav node; delnode external l,r,rtarg;
dleft: (while (l(node) ne Ω and tl l(node)) node=hd l(node);;
right: next=r(node); l(node)= Ω; r(node)=Ω;
if hd next eq rtarg then return;;  <node,x>=next; go to if x
   then dleft else right; end deltrav;
```

To thread an unthreaded tree, we first let $\prec_{tq}$ be the left-node-right traversal order, as defined by a previous algorithm. Then use the following straightforward code, in which *tree* denotes the set of all nodes in the tree:

```
suc={<seq(n),seq(n+1)>,1≤n<#seq}; pred={<tl x,hd x>, x ∈ suc};
(∀n∈tree) l(n)= if l(n) ne Ω then <l(n),t>
   else if pred(n) ne Ω then <pred(n),f> else Ω;
        r(n)= if r(n) ne Ω then <r(n),t>
   else if suc(n) ne Ω then <suc(n),f> else Ω; end ∀n;
```

It is simpler to convert a threaded tree to an unthreaded one, as follows.

```
(∀n ∈ tree) l(n) = if(if l(n) eq Ω then f else tl l(n))
                   then hd l(n) else Ω;
            r(n) = if(if r(n) eq Ω then f else tl r(n))
                   then hd r(n) else Ω; end ∀n;
```

SETL can be used to express algorithms embodying various optimizations. Here, for example, is a parsimonious method, due to Schorr and Waite, for traversing a binary tree; as distinct from the methods given earlier, it avoids the use of a recursion stack. The idea is this: as one descends down a chain of branches to traverse the tree, one reverses the pointers, to get a chain of pointers allowing subsequent ascent. During ascent, the pointers are repaired. We mark those nodes n such that r(n) is the parent of n; in a machine-level implementation, at most one bit is needed for this mark.

```
seq=nl; mark=nl; node=top; par=Ω;
dleft: (while l(node) ne Ω) <node,l(node),par>
            = <l(node),par,node>;;
seq(#seq+ 1) = node;
up: if par eq Ω then go to done;
    if mark(par) then <node,par,r(par)>
            = <par,r(par),node>; mark(node)=Ω;
    go to up;
    else <node,par,l(par)> = <par,l(par),node>; end if;
    if r(par) eq Ω then go to up;;
    <node,r(node),par> = <r(node),par,node>; mark(par)=t;
     go to dleft; {done:} ...
```

3. Sorting.

Occasionally, given a set s and a numerical function f defined on s, one wishes to sort the elements of s according to increasing values of f. The following statement assigns an element of s its position n(s) in sorted order.

```
(∀x∈s) place(x) = #{y∈s | f(y) lt f(x) or
                    (f(y) eq f(x) a place(y) ne Ω)};;
```

More plausible sorting algorithms may also be represented in SETL. Here is the slightly better *insertion sort* which sorts an elements, in place, in a number of steps proportional to n^2.

```
(1<∀j ≤ #seq) k=j-1; (while k gt 0 and seq(k+1) lt seq(k)
                      doing k=k-1;)
<seq(k),seq(k+1)> = <seq(k+1),seq(k)>;;
```

Here is the *bubble sort*, a method about as good as the preceding.

```
n=1; (while n lt #seq) if seq(n+1) gt seq(n) then n=n+1; else
<n,seq(n),seq(n+1)>=<if n gt 1 then n-1 else 2, seq(n+1),seq(n)>;
end while;
```

The simple insertion sort algorithm described above will operate most efficiently if presented with data actually in order. The *sliding insertion sort* or *shell sort* explores this fact, incorporating a device which causes an array to be sorted to converge rapidly to approximate order. The algorithm is as follows. A descending sequence $M_n, M_{n-1}, \ldots, M_1$ of integers is chosen, with $M_1 = 1$. Successively, for each j from n to 1, the array $\{a_k\}$ to be sorted is divided into M_j subsequences

$$a_1, a_{M_j+1}, a_{2M_j+1}, \ldots$$

$$a_2, a_{M_j+2}, a_{2M_j+2}, \ldots .$$

Each of these arrays is sorted using the ordinary insertion sort algorithm. Note that when j = 1 we have $M_1 = 1$, so that the ordinary insertion sort algorithm is eventually applied to

the whole array, guaranteeing that complete order is eventually obtained. Experience shows that it is advantageous to define the sequence $M_n, M_{n-1}, \ldots, M_1$ as follows: put $M_n = 2^k - 1$, where 2^k is the largest power of 2 not exceeding the number of elements to be sorted, and then put $M_{j-1} = M_j/2$. In SETL, all this may be written as follows:

```
m=1; (while m le # seq) m=2*m; end while; m=m-1;
(while m gt 0 doing m=m/2;)(1 ≤ ∀j ≤ # seq) k=j-m;
(while k gt 0 and seq(k+m) lt seq(k) doing k=k-m;)
<seq(k),seq(k+m)>=<seq(k+m),seq(k)>; end ∀j; end while m;
```

Next we describe the so-called *tree insertion sort*. In this method a binary tree, to whose nodes the elements to be sorted are attached, is built up by attaching successive branches. The tree is built in such a way as to ensure that if an element x is attached to a particular node N, then x exceeds all the elements attached to the left-hand sub-tree of N, and is exceeded by all the elements of the right-hand sub-tree of N. The rule for attachment of a new element x is as follows. Examine successive nodes y, beginning at the tree root. If x exceeds y, move down the tree to the right; or if y has no right descendant, make x the right descendant of y. If y exceeds x, move down the tree to the left, or if y has no left descendant make x the left descendant of y. When all the elements of the array to be sorted have been attached to the nodes, 'linearize' the tree to an array by the following recursive rule: first linearize the left hand sub-tree; then take the element attached to the tree root; then linearize the right hand sub-tree to get the top part of the sorted array. This procedure will on the average sort an array of n elements in a tree proportional to n log n. In SETL, it appears as follows.

```
r=nl; l=nl; ntop=newat; elt(ntop)=seq(1);
(1 < ∀j ≤ #seq) x=seq(j); top=ntop;
down: targ=if x gt elt(top) then r(top) else l(top);
      if targ ne Ω then top=targ; go to down;;
      if x gt elt(top) then r(top)=newat; elt(r(top))=x;
         else l(top)=newat; elt(l(top))=x; end else; end ∀j;
seq=nl; traverse ntop;
define traverse top; external seq,elt; if top eq Ω then return;;
traverse l(top); seq(#seq+1)=elt(top); traverse r(top);
return; end traverse;
```

The *simple selection* sort is as follows: survey the n elements of a set to be sorted to find the minimum element; remove it from the array; and iterate. In SETL:

```
sorted=nt; (while set ne nl) sorted(#sorted+1)=[min:xcset]x;
set=set less sorted(#sorted); end while;
```

A variant of this basic idea yields the much faster *tree selection sort*, which may be described as follows. First, attach the elements of the array to be sorted as the twigs of a binary tree of appropriate size. Next, propagate values up to the tree root, attaching to each node the minimum of the values attached to its immediate descendants, and causing each node to point to that immediate descendant node to which this minimum value is attached. When this structure is built, it becomes trivial to locate the original array minimum, detach it from the tree, and move it to a workspace in which sorted array elements are accumulated. After this removal operation, the tree is repaired by redetermining a minimum-of-descendants value for all nodes above the node just removed; after which the selection, removal, and repair process iterates until completion.

Adopting the convention that the minimum-of-descendants is always found down the left-hand branch, the following shows a way that the above algorithm may be written in SETL.

```
/*first build the tree */ l=nl; r=nl;
v = {<newat,seq(j)>, 1 ≤ ∀j ≤ #seq}; trees=hd(v),
loop: newtrees=nl; (while trees ne nl) nd=newat; in from trees;
      rm from trees;
      <l(nd),r(nd),v(nd)>=<ln,rm,if rm ne Ω then v(ln) min v(rm)
                                    else v(ln)>;
      if rm ne Ω then if v(ln) gt v(rm) then
                  <l(nd),r(nd)>=<r(nd),l(nd)>;; end if rm;
      end while; if #newtrees gt 1 then trees=newtrees;
      go to loop;;
/* then put in parent links */ par=nl;
(∀x∈hd(l)) par(l(x))=x;; (∀x∈hd(r)) par(r(x))=x;;
/* now tree is built; begin main selection and repair process*/
top= newtrees; seq=nl;
(while l(top) ne Ω) node=top;
(while l(node) ne Ω) node=l(node);; seq(#seq+1)=v(node);
l(par(node))=Ω; (while par(node) ne Ω)
node=par(node);
if l(node) eq Ω and r(node) eq Ω then l(par(node))=Ω;
   else if r(node) eq Ω then v(node)=v(l(node));
   else if l(node) eq Ω then <v(node),l(node)>
                  = <v(r(node)),r(node)>; r(node)=Ω;
   else if v(l(node)) gt v(r(node)) then
      <l(node),r(node)>=<r(node),l(node)>; v(node)=v(l(node));
   else v(node)=v(l(node)); end if; end while; end while;
```

The still more remarkable *heapsort* remedies certain of the deficiencies of the tree selection sort, and provides a method for sorting an array in place and in a number of steps bounded by n log n. It is in essence a binary tree selection sort in which the tree points are implicit, the descendants of the element at array location j being the elements at locations 2j and 2j+1. The algorithm is as follows.

```
(1 < ∀n ≤ #seq) m=n; (while n gt 1 and seq(n/2) gt (seq(m))
<m,seq(m),seq(m/2)>=<n/2,seq(n/2),seq(m)>;; end ∀n;
(#seq ≥∀top >1) <seq(1),seq(top)>=<seq(top),seq(1)>; m=1;
(while 2*m lt top doing m= targ;)
targ=if seq(2*m) lt seq(2*m+1) and 2*m+1 lt top then 2*m+1
      else (2*m);
if seq(m) lt seq(targ) then <seq(m),seq(targ)>
    = <seq(targ),seq(m)>; else quit;;
end while; end ∀top;
```

Quicksort is a high-speed descendant of the simple bubble sort. It operates as follows: take the first element x of an array a and, as in the bubble sort, compare it to its successor y, interchanging x and y whenever x exceeds y. However, if y exceeds x, interchange y with the element z having the largest possible index consistent with the assumption that x exceeds z. As this process proceeds, an increasing region R_- of elements known to be less than x will build up below x, and an increasing region of elements R_+ known to exceed x will build up above x. Eventually x will come into its proper place. Then, if either R_- or R_+ contains just two elements, they may be placed in order by a single interchange; in the contrary case, the procedure just described may be used recursively to sort R_- and R_+.

In SETL, quicksort appears as follows:

```
define quicksort(a,i,j); if j le i then return;; if j eq i+1 then
if a(j) lt a(i) then <a(i),a(j)>=<a(j),a(i)>;; return; end if j;
bot=i; top=j; (while bot lt top) if a(bot) gt a(bot+1) then
<bot,a(bot),a(bot+1)>=<bot+1,a(bot+1),a(bot)> else
<top,a(top),a(bot+1)>=<top-1,a(bot+1),a(top)>;; end while;
quicksort(a,i,bot); quicksort(a,bot+1,j); return; end quicksort;
```

Merging procedures of various kinds play a central role in many of the most important methods for sorting large arrays by using tapes. Fast internal sorts can also be built using merge techniques. We shall describe one such sort, the so-called

natural two-way merge. It works as follows: given an array of elements to be sorted, use a workspace of equal size, and merge elements from the top and bottom of the array into the bottom of the workspace as long as these elements may be used to form an increasing sequence or *run*. Naturally, if both the top and the bottom element can be used to continue a run, we first use whichever is smaller. When a run cannot be continued, we start a new run, placing it in reverse sequence of positions at the top of the workspace, until once more the run can no longer be continued. At this point, start a new run, storing it at the bottom of the remaining workspace area, etc. When the whole of the original array has in this way been transformed to the workspace, interchange the array and workspace, and repeat. During this process, the number of separate runs into which the total array is divided will be cut in half each time the whole array is combed through, and eventually complete order will result.

In SETL, this procedure appears as follows.

```
start: bot=1; top=#elt; xbot=1; xp=top; flag=-1; extra=nl;
          extra(xbot)=elt(xbot);
onward: if flag eq -1 then xtop=xp; xp=xbot; else xbot=xp;
          xp=xtop; end if;
        flag = -flag;
        (while top ge bot doing xp=xp+flag;)
        cond=if elt(top) ge extra(xp) and elt(bot) ge elt(top)
                then 1
             else if elt(bot) ge extra(xp) then 2 else 0;
        go to {<0,endrun>,<1,tops>,<2,bots>}(cond);
tops:   extra(xp+flag) = elt(top); top=top-1; continue;
bots:   extra(xp+flag)=elt(bot); bot=bot+1; continue; end while;
endrun: if top ge bot then go to onward;;
        elt=extra; if xtop lt # seq then go to start;;
```

Distribution or *pocket sorts* are, in a certain sense, dual to merging sorts. The *simple pocket sort* is the method used to sort punched cards on electromechanical equipment. The

algorithm is as follows. A given collection of keys is to be sorted. One regards these keys as integers to some base p, i.e, as sequences $d_1 \ldots d_k$ of base p digits. The items to be sorted are then distributed into p piles or pockets, according to the least significant digit d_k. Then the piles are gathered up into a single sequence, being taken in the order 0,1,...,p-1, and the distribution process repeated, first for the digit d_{k-1}, then, after regathering, for the digit d_{k-2}, etc. The relative positions assigned in this method to two items during the j-th distribution pass will not subsequently change except as required upon examination of more significant key digits during a later pass; and thus fully sorted order must emerge when all the digits of the keys have been processed.

We may represent this algorithm in SETL as follows.

```
multi=t; q=1; (while multi doing q=q*p;)
   seq=gather(p,dist(seq,p,q,multi));;
definef dist(seq,p,q,multi); pocket=n ;
   (1 ≤ ∀k ≤ #seq) key=(seq(k)/q)//p;
pocket(key, #pocket(key)+1)=seq(k);;
multi= 1 ≤ ∃ k<p|pocket{k)ne nl;
   return pocket; end dist;
definef gather(p,pocket); seq=nl;
(0 ≤ ∀k< p, 1 ≤ j ≤ #pocket(k)) seq(#seq+1)=pocket(k,j);;
   return seq; end gather;
```

The *radix exchange sort* is another fast key sort. It works as follows. Regard each item of the array to be sorted as a boolean string. On a first pass through the array, and by performing appropriate exchanges, place all elements whose lead bit is zero at the bottom of the array, and all elements whose lead bit is one to the top of the array. Then, recursively, apply the same procedure to the top and bottom of the array and to the second through last bit of each item.

In SETL, letting b be the number of bits in a key, this procedure appears as follows.

```
radsort(seq,1,#seq,b);
define radsort(seq,bot,top,b); i=bot; j=top;
(while i lt j) b1=b bit seq(i); b2=b bit seq(j);
if b1 eq 1 a b2 eq 0 then <seq(i),seq(j)>=<seq(j),seq(i)>;
<b1,b2>=<b2,b1>;; if b1 eq 0 then i=i+1;;
     if b2 eq 1 then j=j-1;; end while;
if i-1 gt bot then radsort(seq,bot,i-1,b-1);;
if j+1 lt top then radsort(seq,j+1,top,b-1);; return;
definef b bit word; end=len word -b+1;
   return 1 first (end lant word);
end radsort;
```

The *Ford-Johnson Tournament Sort* reduces, to a level very close to the theoretical minimum, the number of comparisons required to sort n elements. However, the number of moves required will on the average be proportional to n^2. This method is therefore of interest only in the unlikely but conceivable special case in which the cost of comparing items is so high relative to the cost of moving them that attention really does focus exclusively on comparisons. The algorithm is as follows. Divide the items to be sorted into n/2 pairs, arranging each pair so that its first element exceeds its second. Then sort the pairs according to their first elements, using the tournament sort procedure (recursively). This produces a pair of arrays $a_1,\ldots,a_m$ and $b_1,\ldots,b_m$, where $m = n/2$, where the a's are in increasing order, and where $a_j \geq b_j$ for all j. Finding the proper position of one new element among p already ordered others involves q comparisons, where q is the smallest integer such that $2^q-1 \geq p$; and such a location process is at its most efficient when p is precisely of the form 2^q-1. So then

b_1, a_1, a_2 form an ordered sequence of 3 elements, into which b_3 may be inserted using two comparisons;

The set of elements b_1, a_1, and b_3 in proper position are then 3 in number, and b_2 may be inserted among them using two comparisons;

The set of b_1, a_1, a_2, a_3, a_4, together with b_2 and b_3 in proper position, are then seven in number, so that first b_5, and then b_4 may be inserted into position using three comparisons.

$b_1, \ldots, b_5$, together with $a_1 \ldots a_{10}$, are then 15 elements, and thus b_{11}, and then $b_{10}, \ldots, b_6$ may be inserted into position using four comparisons,

and so forth.

This rather complex sorting algorithm may be written in SETL as follows; we assume a set *items* is given for sorting, on which a value-assigning function *valf* having numerical values is defined.

```
definef fordj items; external valf;
if #items eq 1 then return {<1,items>};
else if #items eq 2 then item1 from items; item2 from items;
if valf(item2) lt valf(item1) then <item1,item2>=<item2,item1>;;
   return{<1,item1>,<2,item2>}; end if #; xtra=newat;
map=nl; items2=nl;(while items ne nl) item2 from items;
     item1 from items;
     if item1 eq Ω then item1=xtra; if
(if item1 eq xtra then f else valf(item2) lt valf(item1))
     then<item1,item2>=<item2,item1>;;
item2 in items2; map(item2)=item1; end while;
seq=fordj items2; oseq=[<n,map(seq(n))>, 1≤n≤#seq];
if oseq(#oseq) eq xtra then oseq(#oseq)=Ω;;
oseq(1) insert 1; jbot=2; jtop=3; nelts=3;
(while oseq(jbot) ne Ω doing <jbot,nelts>=<jtop+1,2*nelts+1>;
     jtop=nelts-jtop+1;)
```

```
/* at start of next cycle of comparison, all elements
   b(1) through b(jtop) will have been inserted; the first
   following insertion is of b(k+1), where b(1)...b(jtop)
   and a(1)...a(k) together are nelt elements */
c=jtop ≥ j[j] ≥ jbot | oseq(j) ne Ω;
(while j ge jbot doing j=j-1;)oseq(j) insert(oseq(j) place nelts);;
end while oseq; return seq;
definef elt place nelts; external seq, valf; bot=1; top=nelts;
(while(top-bot) gt 1) mid=top+bot/2;
if valf(seq(mid)) gt valf(elt) then bot=mid; else top=mid;;
   end while;
v=valf(elt); return if v lt valf(seq(bot)) then bot else
   if v lt velf(seq(top)) then top else(top+1);end place;
define elt insert place; external seq;
seq={<if n lt place then n else(n+1),seq(n)>} with <place,elt>;
   return;
end insert; end fordj;
```

4. A first compiler algorithm.

In the pages subsequently to follow we will use SETL to describe a number of algorithmic processes basic to the compilation of programming languages. We now begin by describing a class of *lexical scanners*. These are programs, normally belonging to the very first stages of a compilation process, which accept an input string and break it up into separate *tokens*, i.e., strings of one or more characters representing words of a language. Each token, as it is calculated, is classified according to type (e.g., name, integer, character constant, boolean constant, etc.); basic input conversions, as for example the conversion of a character string representing an integer to the internal form of the integer, may also be performed.

Because of its significant influence on the overall efficiency of the first stages of processing, one normally desires a lexical scanner to be quite fast. For this reason, lexical scanning is customarily performed by a programmed *finite-state automaton* which, driven by an incoming sequence of characters, undergoes a sequence of state transitions until a 'token end' state is reached; then any necessary conversions are performed and a token is emitted. We shall describe a lexical analyzer of this kind. The following background facts should be borne in mind.

i. The states of the automaton correspond to states of uncertainty concerning the nature of the token being constructed. The initial state when a new token is started, called nxt in the formal algorithm below, is one of complete uncertainty. As characters are received, the state of uncertainty will change, always diminishing; when a token-end state is reached, the type of the token is entirely known, and is determined by the final condition of the automaton.

ii. The whole alphabet of characters belonging to a language consists, for the purposes of lexical scanning, of a relatively small number of classes (e.g. alphabetics, numerics, separators, alphabetics having special significance, etc.).

A function *type (character)* is therefore employed by the lexical scanner, which uses the value of this function and its own *state* to find an *action table entry* which describes the action to be taken next. In the following algorithm, the allowed actions are as follows:

a. *end* - end the present token without adding any additional characters to it, and return a triple defining the lexical type of the token, the token itself, and, if applicable, any token associated value to the program calling the lexical scanner.

b. *cont* (continue) - add the current character to the token under construction, and advance to the next input character;

c. *skip* - advance to next input character;

d. *go* (change state) - change to a specified state and then continue as in b);

e. *do* (perform auxiliary process) - perform a specified block of statements, supplied by the programmer. This code may examine and modify the token under construction, the state of the finite state automaton, the *action* parameter (see below) which the lexcial scanner uses, any of its pointers, etc.; if it is some sort of conversion routine, it may supply token-associated data to the lexical scanner. When return from an initial auxiliary code passage is made, a specified sequence of additional actions may be performed. These may include additional auxiliary process invocations, as well as actions of any of the standard types a), b), and c) above. In the algorithm which follows, the auxiliary routines are all part of a common programmer-supplied *auxiliary process package* called *xpak*.

iii. The function which follows is called *nextoken*; when called, it supplies the next following token formed out of a given character string. The routine *setup*, called initially in *nextoken*, builds all necessary tables, and initializes the character string. The form assumed for the action table entries is as follows:

aa. one of the keywords *end*, *skip*, and *cont*; or

bb. an ordered pair <*go*,*statename*>, where *statename* is the name of a lexical analyzer state; or

cc. an n-tuple <*do*,*routname*,...>, where *routname* is the name of an auxiliary routine, and where subsequent components are either *end*, *cont*, *go* followed by *statename*, or *do* followed by *routname*.

The detailed form of our lexical analyzer is as follows:

```
definef nextoken;
  initially setup(typ,table,xpak,cstring); n=1;
  <nxt,end,go,skip,cont,do>= ;;
state=nxt; nn=n-1; data=Ω; token=nulc;
loop: nn=nn+1; action=table(state,type(cstring(nn)));
switch: go to {<end,endc>,<go,goc>,<skip,loop>,<cont,contc>,
              <do,doc>}(hd action);
goc:    state=tl action;
contc:  token=token+cstring(nn); go to loop;
endc:   n=nn; return if data ne Ω then<state,token,data>else
                                        <state,token>;
doc:    <-,rout,action>=action; xpak(rout);
        go to if action eq Ω then loop else switch;
end nextoken;
```

This routine is simple enough; as we shall soon see, the routine *setup* which supplies the tables needed by *nextoken* is rather more complex. Concerning *setup*, we have made the following assumptions.

i. It finds the information needed to define the character type function *typ*, the action table *table*, and the comprehensive package *xpak* of auxiliary processes at the head of the system input string, all represented in a form able to be read using the SETL *read* statement. This information is read, checked for accuracy, and converted appropriately to produce the required tables.

ii. In more detail, the information supplied to *setup* is as follows. First, a sequence

[ctype1,...,ctypen]

in external form (i.e., in a form suitable for ingestion by a *read* statement), defining the full collection of character types with which the lexical scanner will be concerned.

Suppose, for example, that we consider a hypothetical language lexically somewhat like FORTRAN, in which the allowed lexical types are as follows:

a. Integer: any sequence of digits, embedded blanks allowed.

b. Real number: an integer, followed by a decimal point, and optionally followed by a second integer.

c. Name: any string of nonspecial characters beginning with an alphabetic; no embedded blanks allowed.

d. Special character: any character other than blank or period.

e. Period delimited operator: any string of nonspecial characters, beginning with an alphabetic, containing no blanks, and delimited fore-and-aft with a period. Examples would be: .ge. , .shift. .

f. Hollerith constant: any number of digits, followed by the letter H, followed by an arbitrary character string of the length specified by those two digits. An example would be: 5hhooka. We suppose for simplicity that *eR* functions as an end-of-statement signal, no continuation-card feature being provided.

For such a language, the relevant character types could be declared to *setup* in the following form

(1) [a,h,'1',+,.,bl,er] .

Here 'a' is used to designate the type of the typical alphabetic (which is to say alphabetics not equal to h, since h plays a special role in hollerith constants); 1 to designate the type of a numeric; + to designate the type of a special symbol other than '.', etc. We use 'er' to designate the type of an end record symbol, 'bl' to designate the type of a blank.

iii. Next there follows a set of ordered n-tuples, which together define the lexical type of every possible character. These n-tuples have the form

$$\langle type, cstring_1, cstring_2, \ldots \rangle$$

where *type* is a previously declared lexical type, and $cstring_j$ is a string of characters, all of which are declared to have this type. The special character string 'er' is however reserved to represent the SETL end record character ↵.

In the case of our hypothetical FORTRAN-like language, we would have

(7) {<a, abcdefgijklmnopqrstuvwxyz>,<'1','0123456789'>,

<+,'()[]{}*+-/=<>≤≥∩$@?∃:|',' >,

<.,'.'>,<h,h>,<er,'er'>,<bl,' '>}

iv. Next there must follow a set of ordered pairs serving to define the action table of the lexical processor. Each of these pairs has the form

$$\langle state, [aent_1, aent_2, \ldots, aent_k] \rangle .$$

Here, *state* is a state of the lexical scanner, while each *aent* is an action table entry, having one of the allowed forms described above. The number of *aent* terms shown in the sequence displayed above must equal the number of character types declared; the j-th *aent* term will be consulted when a character of the j-th type is encountered and the lexical scanner is in the specified *state*.

In the case of our hypothetical FORTRAN-like language, we might employ the following lexical states (which, as have already been noted, corrasponded to the various states of uncertainty which could arise as we progressively scanned a token from left to right;

nxt - a new token is just starting;
nm - a name is being scanned;
irh - an integer, a real number, or a hollerith constant is being scanned, but we are still not sure which;
ir - having encountered a blank in a string of digits, we are sure that an integer or a real is being scanned;
dip - having encountered a period, we are scanning either a real or an integer followed by a period delimited operator;
r - definitely scanning a real number;
pd - definitely scanning a period-delimited operator.

In the case under consideration, our action table would be described as follows, recalling that corresponding character types are [a,h,'l',+,.,bl,er]);

(3) {<nxt,[<go,nm>,<go,nm>,<go,irh>,<do,spend,end>,
<go,pd>, skip, <do,erend,end>]
<nm,[cont,cont,cont,end,end,end,end]>,
<irh,[end,<do,holcon,end>,cont, end, <go,dip>,<go,ir>,end]>,
<ir,[end,end,cont,end,<go,dip>, skip,end]>,
<dip,[<do,back,end>,<do,back,end>,<go,r>,end,end,<go,r>,end]>,
<r,[end,end,cont,end,end,skip,end]>,
<pd,[cont,cont,cont,end,end,end,end]>}

v. Next must follow text defining every auxiliary process mentioned in the state table. This text is supplied as a set of ordered pairs, each having the form

<*name*,*text*> ,

where *name* is the process name, and *text* is its body. This collection of pairs is converted to a complete body of code having the form

$name_1$: $text_1$ return;

$name_2$: $text_2$ return;

...

together with calculated go-to statement which, supplied with an identifier, invokes an appropriate auxiliary process.

The auxiliary routines may refer to the lexical analyzer's 'beginning of present token' pointer as *tokbegin*, its 'current symbol' pointer as *curpointer*; the token being formed as *token* and the scan state as *state*. Note that *nextoken* returns *state*, *token*, and *data* to the main program upon encountering an *end* command.

The value of *state* when the lexical action *end* is executed therefore defines the lexical type of *token* to the main program. In certain cases where an *end* is to be executed forthwith, our auxiliary routines may therefore set *state* to values not appearing in the action table. This is merely to signal the main program. When *nextoken* is called again, *state* will always be set to *nxt*. Note as an example of this that in the package of auxiliary procedures which follows, the state 'er' indicates end of current record reached, and 'ef' indicates end-of-file.

Considering our hypothetical FORTRAN-like language once more, and noting the occurrences of auxiliary process names in the action table description given above, we would supply the following auxiliary processes.

```
(4) {<spend,'token=cstring(curpointer); curpointer = curpointer+1;'>,
    <eread,'if cstring(curpointer+1) eq er then curpointer=curpointe
       state = 'ef'; token=er+ef; else state = 'er'; end if;'>,
    <back,'curpointer = curpointer-1;'>,
    <holoon, 'n=dec token; curpointer = curpointer+2, token=nulc;
    (1 ≤ ∀j ≤ n) if cstring(curpointer) ne er then
       token=token+cstring(curpointer); curpointer=curpointer+1;
       else quit;; end ∀j;'>) .
```

Note that (1), (2), (3), and (4) together constitute a complete description of a FORTRAN-like lexical scan. Of course, from another point of view, the text of *nextoken* and of the associated routine *setup* must also be reckoned as part of this description.

It is interesting to use the mechanism just described to describe the lexical structure of SETL. In the version of SETL actually to be keypunched, underlined characters will not be available; we will therefore represent every underlined name by *period-terminated name*, identical with the word underlined except for the absence of underling and the presence of a terminating period. Thus, for example, we will keypunch

t, nl, nulc, with

as

t., nl., nulc., with.

respectively. This being the case, keypunched SETL has the following lexical elements:

names, e.g. setname, sl
period delimiated names, e.g. with., opl.
special characters
integers
character string constants (in quotes)
bit-string constants, designated either with a b or with an

A lexical description of a plausible keypunched version of SETL is then as follows.

```
[a,o,b,1,7,$,+,.,qt,bt,er]
{<a,acdefghijklmnpqrstuvwxyz>,<o,o>,<b,b>,
<1,'01'>, <7,'2345G7'>, <$,'89'>
<+,'()[]{}*+-/=<>&0$¢¥3:|,'>, <qt,''''>,
<.,.>, <er,er>, <b ,' '>}
{<nxt,[<go,npd>, <go,npd>, <go,npd>, <go,numbit>, <go,numoct>,
      <go,num>, <do,spend,end>, <do,spend,end>,
      <do,skip,go,cs>, skip, <do,cer>]>,
<npd,[cont,cont,cont,cont,cont,cont,end, <do,pd,end>,end,end,end
<cs,[cont,cont,cont,cont,cont,cont,cont,cont, <do,qtest>,
      cont,<do,cer>]>,
```

```
<numbit,[end,<do,cmake,end>, <go,bitoct>, cont, <go,numoct>,
    <go,num>, end, end, end, skip, end]>,
<bitoct,[end,end,end,cont,cont,end,end,end,end,skip,end]>,
<numoct,[end,<do,cmake,end>,end,cont,cont,<go,num>,
    end,end,end,skip,end]>,
<num,[end,end,end,cont,cont,cont,end,end,end,skip,end]>]
[<spend,'token=cstring(curpointer); curpointer=curpointer+1;
    state="special";'>,
<pd, 'token=token+".";curpointer=curpointer+1; state="pd";'>,
<gtest, 'curpointer=curpointer+1; action=if cstring(curpointer)
    eq "" then "cont" else "end";'>,
<cer, 'action=if cstring(curpointer+1) eq er then "end"
    else "skip";'>,
<cmake, 'state="oct";'>,
```

The following remarks will aid the reader in examining this material. The lexical states involved are as follows:

npd - scanning a name or a period-delimited operator;
cs - scanning a quoted character string;
numbit-still within an initial sequence of zeros or ones, which might either be part of a pure bit-string constant, an octal constant, or an integer;
bitoct-having passed a 'b', definitely within a bit string or octal constant;
numoct-having passed a digit in the range 2-7, definitely within either an octal constant or an integer;
num - definitely within a decimal integer

Recall that *state* will be returned by *nextoken* to the main program. Additional values assigned by *spek* to *state* which will be returned by *nextoken* to the main program include 'er' which means end of current record; 'ef' which means end of current file, 'special', which indicates a special symbol; and 'oct', indicating that token is an octal number. The correspondence between character types and lexical notions will be easier to spot if the action table is written out on a large sheet of paper as a rectangular array.

After all these preliminaries, we shall now give the SETL code for the lexical *setup* routine. This code has, in a miniature way, many of the features associated with larger compilers. It is in fact a compiler of sorts, transforming tables like those shown above into tables directly interpretable by the rather simple *nextoken* algorithm. Typically enough for programs of this kind, much of *setup* is concerned with verification of the correctness of the data presented to it, and with the printing of diagnostics where required. Aside from this and from some rather straightforward transformation of table form, the principal responsibility of *setup* is to build up text for the routine *rpak*, and to use the SETL dynamic compilation feature to produce this routine. The assumed form for the *rpak* text is as follows.

```
define rpak(numrout);
nextoken external cstring, tokbegin, curpointer, state,
     token, data;
go to {<1,rout1>, <2,rout2>, ...} (numrout)
rout1: text1     return;
rout2: text2     return;
     ...
     end rpak;
```

Here, $rout_j$ is the j-th auxiliary procedure name supplied by the programmer, and $text_j$ is the text defining this procedure. Note in what follows the useful function *is*: (a is b) assigns b to a and has value b. This provides a convenient way of identifying parts of expressions within *setup*; *errs* is a set of possible errors, which will be printed if not empty. Here finally is the detailed *setup* code itself.

```
define setup(typ,table,rpak,cstring)
nextoken external (n,tokbegin),(no,curpointer),state,token,data;
initially allc='abcdefghijklmnopqrstuvwxyz0123456789'+
          '(){}[],*+-/=<><>Ω∃#∀∃:.|'''+
          er; allc={j elt allc, 1 ≤ j ≤ len allc};
          nxt=; end initially;
error=f; read seqtypes;  do readcheck (seqtypes);
block readcheck(x); if x eq Ω then print
   'run terminated by illformed set representation.';
   exit;; end readcheck;
read ctypes; do readcheck(ctypes); (∀tup ε ctypes)<type,clist>=tup;
   (while clist ne Ω) <cstring,clist> = clist;
   if cstring= 'er' then do setype(er) else
   (1 ≤ ∀j ≤ len cstring) c=j elt cstring; do setype(c);; end if;
   end while clist; end ∀tup;
block setype(c); typef{c}= typef{c} with type; end setype;
/*check that range of typef are elements of sequence of types*/
if(ers is  {ty ε tℓ [seqtypes]|n ty ε tℓ[typef]}) ne nℓ then
   print 'types specified but not used are: '',ers; do e;;
block e; error=t; end e;
if(ers is {ty ε tℓ[typef]|n ty ε tℓ[seqtypes]}) ne nℓ then
   print 'unspecified types are used; these are: ',ers; do e;;
/*check that all characters have unique type specified*/
if(ers is {c ε allc | typef{c} eq nℓ}) = ne nℓ then
   print 'type unspecified for following characters: ',ers; do e;;
if(ers is {c ε allc|(#typef{c}) gt 1}) ne nℓ then
   print 'type multiply specified for following characters:',ers;do
typ=typef; read rawtable; do readcheck(rawtable);
   statesused= hd[rawtable];
/*check that 'nxt' belongs to statesused, and that there are
   no repetitions */
if n 'nxt' ε statesused then
   print 'required state "nxt" omitted from table'; do e;;
if (ers is{st ε statesused | (#rawtable{st}) gt 1}) ne nℓ then
   print 'multiply defined states: ',ers; do e;;
```

```
/* force to single valued function */
  (∀x ε statesused) rawtable(x) = arbitrary rawtable(x);;
/*check that right number of terms in all sequences*/
if(ers is{st ε statesused|(#rawtable(x)) ne #seqtypes}) ne nl then
  print 'states defined with wrong number of type entries:',ers;do e;;
/*convert to two dimensional table*/
table = {<state, seqtype(j), rawtable(state)(j)>,
      state ε statesused, 1 ≤ j ≤ #seqtype}
/* check that all nonpair entries are either end, skip, or cont*/
if(ers is{<x,y,table(x,y)>,x ε statesused, y ε tl{seqtypes}|
  n pair table(x,y) and n table(x,y) ε {'end','skip','cont'}
  or pair table(x,y) and n (end table(x,y))ε{'go','call'}}) ne nl
then print 'illegal entries in following positions of table:ers;do e;;
/*check that all go-to entries are well-formed*/
if(ers is {<x,y, table(x,y)>, x ε statesused, y ε tl{seqtypes}
  (hd table(x,y)) eq 'go' and pair table(x,y) and
  n tl table(x,y) ε statesused}) ne nl then
print 'illformed go-to entries in following positions of table:
    ', ers; do e;;
/* now prepare to check wellformedness of all call-type entries*/
read routset; do readcheck(routset); routs=hd{routset};
  routscalled=nl;
/* all routines uniquely defined */
if (ers is {rt ε routs|(#routset(rt)) ne 1}) ne nl then
  print 'illdefined or multiply defined routines: ',ers; do e;;
if(ers is{<x,y,table(x,y)>, x ε statesused| ∃[y] ε tl{seqtypes}|
  pair table(x,y) and (hd table(x,y)) eq 'call' and n callok
  table(x,y)}) ne nl
then print 'illegal call-type entries in following positions:',ers;do e;
definef callok entry; setup external routscalled, statesused;
  tup=entry; ok=t;
(while tup ne nl) <key,tup>=tup;
  if key ε {'end', 'skip', 'cont'} then continue;;
  if key eq 'go' then <label,tup>=tup;
  if label eq Ω then return f;
    else if n labelsstatesused then ok=f;; continue; end if key;
```

```
   if key eq 'call' then <rout,tup>=tup;
   if rout eq Ω then return f;
     else rout in routscalled;; continue; end if key; end while;
return ok; end callok;
/*check that all routines called are defined */
if (ers is {rt ε routscalled | n rt ε routs}) ne nl then
   print 'routines used but not defined: ', ers; do e;;
/*give warning diagnostic on superfluous definitions */
if (ers is {rt ε routs|n rt ε routscalled}) ne nl then
   print 'warning *-*-*- routines defined but not used:',ers; do e;;
/* number toutines */
rnums = nl; (∀r ε routs)rnums(r)=#rnums+1;;
/*set up rpak for compilation, compile it, and check success */
paktext='define rpak(numrout); setup external cstring, takbegin,
        curpointer  state, token, data;'+
   'go to {' + [+: rout ε routs] ('<'+ dec rnums(rout)+ ',' +rout
        +if rnums(rout) ne #rnums then ',' else'} (numrout);')
        +[+: rout ε routs](rout+':'+ routset(rout)+'return;')
        +'end rpak;'
rpak=compile paktext; if rpak eq Ω then do e;;
   if error eq t then print 'run terminated by lexical error'; exit;
/* now replace rout names in action table by corresponding
   index in rpak */
else (∀x ε statesused, y ε tl[seqtypes]|(hd table(x,y)) eq 'do')
   j=1; (while op is <* z j> table (x,y)) ne Ω doing j=j+1;)
   if op eq 'do' then
     table(x,y)=<rnums(<*z(j+1)>table(x,y)) z (j+1)>table(x,y);
   end if; end while, end ∀x;
/*now rpak has been compiled, typef supplied, and table constructed*
/*do read-in operation until double end record*/
record(input); this=Ω; cstring=nulc;
(while n(prev eq nulc and this eq nulc)) prev=this;
this=record(input); cstring=cstring+this+er; end while;
return;
definef name is exp; name=exp; return exp; end is; end setup;
```

4. Miscellaneous combinatorial algorithms.

In the next few paragraphs, we will set out various algorithms of a rather mathematical flavor, related in diverse ways to combinatorial situations of theoretical interest. These algorithms are intended to demonstrate the ease with which SETL adapts itself to a variety of combinatorial structures and situations. Our first example is simple but famous: Cantor's 'diagonalizer', which, given a set s and a multivalued map f : s → s, produces a set which is not of the form f(s). It is

diagset = {x ∈ s | ∄ x ∈ f(x)};

the reader may supply the proof!

Next we present some useful "closure" algorithms. If as and bs are subsets of a set s, and f is a (possibly multivalued) map on s; the following sets are often of interest.

close(f,as), all points obtained by repeated applications of f to as; and closure(f,as,bs), all points obtained by repeated applications of f to as, taking only images in bs. The corresponding SETL algorithms are as follows.

```
definef close(f,as); im=f{as}; n=0; (while n lt #im) n=#im;
im=im u f{im};;return im; end close;
definef closure(f,as,bs); im=f{as} int bs; fp=fgcf|hd gcbs and
tl gcbs): n=0; (while n lt #im) n=#im; im=im u fp{im};;
return im; end closure;
```

This algorithm returns f' such that f'{x} = close(f,{x}) is:

```
definef closef(f,set); fp=f; n=0; (while n lt #fp) n=#fp;
(∀s ∈ set) f{s}=f{s} u fp{s};; end while; return fp; end closef;
```

Next we give an algorithm related to *Dilworth's theorem*. The theorem is this: if s is a partially ordered set, i.e.,

a set on which a transitive relationship xRy is defined, then the minimum number n of completely ordered subsets by which s can be covered is equal to the maximum number m of elements which can be found in s, no two of which are related by the relation R. Here we call a subset t of R completely ordered if given any x and y in t, we have either xRy or yRx. This rather abstract-sounding result has interesting applications to parsing! Its proof, which, like the proof of most of the other combinatorial theorems we shall consider, is inductive and goes as follows. Clearly $n \geq m$, so only $m \geq n$ must be proved. Call an element x *maximal* if yRx is false for all $y \neq x$; and *minimal* if xRy is false for all $y \neq x$. If there exists an element x_0 unrelated to any other element, remove it to obtain a smaller set s', in which clearly there can exist no more than m-1 mutually unrelated elements; apply Dilworth's theorem to s', and make x_0 a chain by itself. In the contrary case, find, if possible, a set c of m unrelated elements, such that c does not either consist entirely of maximal or entirely of minimal elements. Then if xRy for one $y \in c$, a situation we will indicate by writing xRc, we can never have $\bar{y}$Rx for a $\bar{y} \in c$ (which we would indicate by writing cRx); for this would imply $\bar{y}$Ry. In this case, use c to divide s into two parts; one with c and cRx, the other with c and xRc. Inductively apply Dilworth's theorem to both parts, and join the resulting chains into longer chains whenever they contain a common element of c. If there exists no such c, find a minimal x and a maximal X such that XRx; then remove X and x from s, obtaining a set s'. If s' contained a set t of m mutually unrelated elements, then clearly they would either all have to be minimal or all maximal elements of s. But then plainly s would either contain m+1 maximal or m+1 minimal, and hence m+1 unrelated, elements. Since this is impossible, we conclude that there exst at most m-1 unrelated elements of s'. Then apply Dilworth's theorem to s', covering it by m-1 ordered subsets; and add the subset {x,X} to these.

This proof may be written as a SETL algorithm, as follows.

```
definef dilworth(set); external rela;
/*this function produces a set of sequences, ordered in the
  sense of the relationship reln, which cover set.  We suppose
  reln(x,y) to have values t, f */
if set eq nl then return nl;;
doms={<x,y>, xεset, yεset | reln(x,y)};
domby = {<tl x, hd x>, x ε doms};
if ∃[x] ε set | doms(x)eq nl and  domby {x} eq nl then
/* remove unrelated element */
 return dilworth(set less x) with {<1,x>};;
maxelts = {x ε set | domby(x) eq nl};
minelts = {x ε set | doms(x) eq nl};
cut := maxun(set);
if cut eq maxelts or cut eq minelts then go to check;;
[cutcase:] top = domby(cut) u cut; bot = dom(cut)u cut;
   dilt=dilworth(top); dilb=dilworth(bot); dil=nl;
  (∀c εdilb) if ∃[d]εdilt| d(1) eq c(#c) then
/*common element, join*/
   (c u {<#c+j-1, d(j)>, 2 ≤ j ≤ #d}) in dil; dilt=dilt less d;
   else /* no common element */ c in dil;
  . . . . . .
 end ∀c;
 return dil u dilt;
[check:]  if #maxelts ne #minelts then go to checkwhich;
  if ∃[x] ε maxelts, [y] ε minelts, [z]ε{maxun(set less x less y)}
        |(#z) eq #cut then cut=z; go to cutcase;;
[nocut:]  /*no c, find minimal mn, maximal mx */
  mn = ∃minelts; mx=∃{x ε domby(y)|x ε maxelts};
  return dilworth(set less mn less mx) u {<1,mn>,<2,mx>};
[checkwhich:] if ∃[x] ε(if #maxelts ≥ #minelts then maxelts
        else minelts), [z] ε {maxun(set less x)}|(#z) eq #cut
        then cut=z; go to cutcase; else go to nocut;
        end if x; end dilworth;
```

```
definef maxun(set); dilworth external reln;
  if set eq nl then return nl;
  return {maxno: x∈set}(maxun({y∈set n(reln(x,y) or reln(y,x))})
        with x); end maxun;
definef a maxno b; return if #a gt #b then a else b; end maxno;
```

Next we consider the so called combinatorial "matching" or "marriage" problem, and an algorithmic version of its solution. The problem is this: given a multivalued map on a set s, when can we find a one-to-one map g defined on s such that $g(x) \in f(x)$? The necessary and sufficient condition is that $\#f[t] \geq \#t$ for each subset t of s. This condition is plainly necessary. The proof that it is also sufficient is inductive. Call a subset t of s *thin* if $\#f[t] = \#t$; otherwise *thick*. If t_1 and t_2 are thin, then

$$\#(f[t_1] \cup f[t_2]) = \#t_1 + \#t_2 - \#(f[t_1] \cap f[t_2])$$

$$\leq \#t_1 + \#t_2 - \#(t_1 \cap t_2) = \#(t_1 \cup t_2),$$

so that clearly $t_1 \cap t_2$ is thin also. Hence, if there exist any thin sets, there exists a minimal thin set t_0. We may prove in much the same way as above that if $x \in t_0$, $y \in f(x)$, and t is thin, then if f[t] contains y, t must contain x. This is the key to the problem, since it enables us to proceed as follows: if s contains no thin subset, remove some element from f[s], and apply induction. If s contains a minimal thin subset, take an element x of it, and an element y of f(x). Then define g(x) = y; remove y from the range of f; and define g on the complement s' of x inductively as a solution of the matching problem on s'.

In SETL we have:

```
definef match(f); if f eq nl then return nl; t=minthin(f);
  if t eq Ω then y= tl ∈f; return match({p∈f|(tl p)ne y});
x=∈t; y=∈f(x); return match(f less <x,y>) with <x,y>;
end match;
```

```
definef minthin(f); s=hd[f]; if 1 ≤ ∃[k] ≤ #s, [t]∈npow(k,s) |
        (↓f[t]) eq ↓t then return t;
  else print 'necessary condition for matching problem violated';
  exit;;end minthin;
```

This construction has a wide variety of interesting extensions. Suppose, to give just one example, that nm(x) is a numerically valued function defined on the domain s of f, and that we are required to construct a multi-valued function g such that g(x) is always a subset of f(x), such that g(x) always contains nm(x) elements, and such that all the sets g(x) are disjoint. (For nm(x) ≡ 1, this reduces to the case that has been considered.) The necessary condition is clearly $\downarrow f[t] \geq \sum_{x \in t} nm(x)$ for all subsets t of s. We may easily see that this condition is also sufficient as follows: replicate each point x∈s, nm(x) times; put $f(\bar{x}) = f(x)$ for each replica $\bar{x}$ of x. Solve the matching problem with nm ≡ 1 for f on this larger domain. This clearly gives a solution to our generalized matching problem. In SETL, this construction may be written as

```
definef genmatch(f,nm); nf=nf;
  replicas = {<x,x,k>, x ∈ hd[f], 1 ≤ k ≤ nm(f)};
  (∀x ∈ hd[f], y ∈ replicas(x)) nf(y)= f(x);;
  return matches(nf) ⊆ replicas; end genmatch;
```

Here we may define the generally useful functional composition mapping as follows.

```
  define f ⊆ g; return{<x,y>, x∈hd[g], y∈f[g(x)]}; end ⊆;
```

We continue this sequence of combinatorial algorithms by discussing the so-called *Menger curve theorem*, which applies to *unordered graphs*. An unordered graph is a set of nodes, together with a symmetric relationship between pairs of nodes, which tells us when a pair of nodes is connected by an edge, i.e., are *neighboring*. We may also think of this as a multi-valued function *maybe*, defined on s, which gives us

the set of neighbors of each node, and which has the property that $x \in nayb_s(y)$ implies $y \in nayb_s(x)$. Given two sets a and b in a graph, we say that they can be *connected* if there exists a sequence of nodes, the first in a, the last in b, every successive pair of nodes being neighbors; such a sequence is called a *path* connecting a and b. We say that a and b are *n-connected* in the graph if, whenever n-1 nodes are removed from the graph, the parts of a and b which remain are still connected. Menger's theorem asserts that if a and b are two disjoint sets in a graph, and if a and b are n-connected in the graph, there exist at least n disjoint paths connecting a and b. The proof is inductive, and uses the matching theorem stated above as a lemma. First consider the case in which a and b together include all the nodes of the graph. If there do not exist n nodes whose removal disconnects a and b, then drop one node from a and proceed inductively. If there does exist such a collection of n nodes, a certain subset of them, call it $\bar{a}$ (resp. $\bar{b}$) will belong to a (resp. b). If $\bar{a}$ contains a subset $\hat{a}$ of k elements which has fewer than k neighbors in the set $b-\bar{b}$, then let $\hat{b}$ be this set of neighbors. Given that in the case we are considering $a \underline{\cup} b$ is the full set of nodes of the graph, it is clear that the removal of a set s of nodes from $a \underline{\cup} b$ will disconnect them if and only if each edge of the graph which connects a and b has at least one end in s. From this it is easy to see that the union of $\bar{a}-\hat{a}$, $\hat{b}$, and $\bar{b}$ disconnects a from b. But this union has at most n-1 points, and we have a contradiction. We conclude that every subset $\hat{a}$ of $\bar{a}$ containing k elements has at least k neighbors in $b-\bar{b}$; thus the matching theorem can be used to define disjoint paths between $b-\bar{b}$ and all the points of $\bar{a}$. Similarly, disjoint paths between $a-\bar{a}$ and all the points of $\bar{b}$ can be found; and altogether we have n disjoint paths between a and b.

Next consider the case in which there exist nodes x not in the union of a and b. Take such an x. If the removal of x still leaves a and b n-connected, remove it and proceed inductively.

In the contrary case, there exists a set c of n points containing x, such that the removal of c from the graph disconnects a from b. We call c a *cut* between a and b. Clearly, every path in the original graph connecting a and b must contain at least one point of c.

Divide c into three parts: $\bar{a}$, the nodes of c belonging to a; $\bar{b}$, the nodes of c belonging to b; and $\bar{c}$, the nodes of c belonging neither to a nor to b. If a point y can be connected to a point of a by a path not passing through any point of c, say that y lies on the a *side* of c; and define the b side of c similarly; designate the a side of c by as, and the b side of c by bs. Clearly as and bs are disjoint. Since both a and b must contain n points, neither as nor bs is null. Thus aa = as $\underline{u}$ $\bar{b}$ $\underline{u}$ $\bar{c}$ and bb = bs $\underline{u}$ $\bar{a}$ $\underline{u}$ $\bar{c}$ are both proper subsets of our original set of nodes; clearly, these sets have only $\bar{c}$ in common. If e-$\bar{a}$ can be disconnected within aa from $\bar{b}$ $\underline{u}$ $\bar{c}$ by the removal of a set e containing fewer nodes than $\#\bar{b}$ + $\#\bar{c}$, then, since every path connecting a-$\bar{a}$ to b must pass through $\bar{b}$ $\underline{u}$ $\bar{c}$, the removal of $\bar{a}$ $\underline{u}$ e from our set of nodes would disconnect a from b. Hence this is impossible; and therefore Menger's corve theorem may be applied inductively to construct $\#\bar{b}$ + $\#\bar{c}$ disjoint paths in aa connecting a-$\bar{a}$ to $\bar{b}$ $\underline{u}$ $\bar{c}$. Similarly, we may construct $\#\bar{a}$ + $\#\bar{c}$ disjoint paths in bb connecting b-$\bar{b}$ to $\bar{a}$ $\underline{u}$ $\bar{c}$. These paths clearly constitute:

i. $\#\bar{b}$ paths connecting a-$\bar{a}$ to $\bar{b}$

ii. $\#\bar{a}$ paths connecting b-$\bar{b}$ to $\bar{a}$

iii. $\#\bar{c}$ paths in aa and $\#\bar{c}$ paths in bb connecting a-$\bar{a}$ to $\bar{c}$.

Every point x in $\bar{c}$ is clearly the endpoint both of a path of type iii lying in aa and a path of type iii lying in bb; and these paths clearly have only x in common. Joining them into a longer path at their common endpoint c, we get from the paths i, ii, and iii a total of n disjoint paths connecting a and b.

This inductive procedure may be written as a SETL routine, as follows.

```
definef menger(s,a,b,n); external naybs, n; if n eq 0 or
        s eq nil then return nil;;
if ∃[x] ∈ s | n s ∉ a and n s ∈ b then go to cutcase;;/*aub not s,
if ∃[nnodes] ∈ npow(n,a u b) | (∀y∈a, z∈b|y∈nnodes or
        z∈ nnodes or n y ∈ naybs(z)) then go to mincase;;
drop= ∞a; return menger(s less drop, a less drop,b,n);
[mincase:] abar = a int nnodes; bbar = b int nnodes;/*use matching*/
mapa = {<x,y>, x∈abar, y∈naybs(x)|y∈b a n y∈bbar}; mapa=match(mapa);
mapb = {<x,y>, x∈bbar, y∈naybs(x)|y∈a a n y∈abar}; mapb=match(mapb);
return {{<1, hd p>,<2, tl p>}, p∈mapa} u {{<2,hd p>,<1,tl p>},p∈mapb};
[cutcase:] if ∃[xnodes] ∈ npow(n-1,s less x)|(∀x∈a side(xnodes with x))
        |n x∈b) then go to mincut; else
  return menger(s less x,a,b,n)  /* a,b still connected, remove x*/;
[mincut:] cut=xnodes with x; abar=a int cut; bbar=b int cut;
  cbar={y∈cut|n y∈a and n y∈b};
  acurves=menger(a side cut u bbar u cbar,{y∈a|n y∈abar},
        bbar u cbar, #bbar+#cbar) /* connect a-ā to b̄uc̄ */;
  bcurves=menger(b side cut u abar u cbar, abar u cbar,
        {y∈b|n y∈bbar}, #abar+#cbar); /*connect b-b̄ to āuc̄ */
  cset={c∈acurves|n c(#c)∈bbar} u {c, c∈bcurves|n c(1)∈abar}
    n{c join d, c∈acurves, d∈bcurves|c(#c)∈cbar and d(1)=c(#c)} ;
definef s1 join s2; return s1 u {<#s1+j-1, s2(j)>, 2≤j≤#s2};
        end join;
return cset;        end menger;
definef a side c; menger external s, naybs;
  return closure(naybs,a,{x∈s|n x∈c}); end side;
```

The algorithms for the Dilworth construction, the matching problem solution, and the Menger construction which have been given are certainly inefficient, and, if they were to be seriously used, would require substantial improvement. The sign of this is the frequent reference made in these algorithms to the very expensive power set generator functions npow. Such imporvement belongs, of course, to the abstract

range of matters expressible in SETL; merely improved coding will normally fail to have a sufficiently strong effect on algorithms of this combinatorial kind. One might expect, for example, that a hand-coded, carefully machine-optimized version of the SETL algorithm for the Dilworth construction could handle sets of n+1 or n+2 nodes as fast as the SETL program itself could handle n nodes. Without an improvement in the strategy of approach, these algorithms, however coded, are prohibitively costly. The point we wish to make however is that the tunes which can be played on a mathematical flute can also be played in SETL, in about the same way; provided of course that the mathematical construction does not involve essential references to infinite sets.

Next we give a set of combinatorial generators of a rather different character. These are all combinatorial *generators*, i.e., algorithms which generate all the members of a certain interesting class of combinatorial objects. The first of these is a program to generate all permutations of n objects, which we give here because of its intrinsic interest, even though most of our discussion of algorithms related to permutations is reserved for a later paragraph.

This program generates all permutations of n in lexical order. Like most of our other generators, it works by iterative application of a rule which defines the k+1-st object to be generated in terms of the k-th object; where the enumeration of objects is to be in some well-defined, generally lexicographic, order. For permutations, we do in fact use the standard lexicographic order. Then the next permutation after a given s_n is defined by the following rule: increase the last possible element by the smallest possible amount. That is, given s_n, find the last element s_j which is not part of a monotone decreasing "tail", interchange it with the smallest s_k with $k>j$ and $s_k>s_j$, and then place all the elements $s_{j+1},\ldots,s_n$ into ascending order. In the program which now follows, a signal is transmitted through "more" when the process restarts.

```
definef perm(n,more);
/*initialize if new*/
if n more than more=t; seq={<j,j>, 1<j<n}; return seq;;
/*if sequence is monotone decreasing, there are no more
  permutations. Otherwise find last point of increase*/
if n (n > ∃[j] ≥ 1|seq(j) lt seq(j+1)) then more=f;
  return Ω; end if;
/* then find the last seq(k) which exceeds seq(j) and swap*/
find = n ≥ ∃[k] > j | seq(j) lt seq(k);
  <seq(j),seq(k)> = <seq(k),seq(j)>;
/*then rearrange all the elements after seq(j+1) into
  increasing order*/
(j < ∀k ≤ (n+j+1)/2) kk=n-k+j+1;
  <seq(k),seq(kk)> = <seq(kk),seq(k)>; end ∀k;
return seq; end perm;
```

A routine for generating the elements of the set npow(n,s) may be based on the same principle. We arrange the elements of s in a sequence, and always arrange each subset of s in the increasing order of this sequence. The rule for obtaining the next subset after a given one is again: increase the last possible element by the smallest possible amount; and reduce all the elements which follow it by the greatest possible amount. In SETL we have:

```
definef nexnpow(n,s,more);
if n more then more=t; seq=nl; (∀x∈s) seq(#seq+1)=x;;
select={<i,i>,1≤i≤n}; return seq{tl{select}}; end if;
if select(n) ne #seq then select(n)=select(n)+1;
      return seq{tl{select}}; end if;
if n (n > ∃[j]≥1 | select(j+1) gt select(j)+1) then
      more=f; return Ω; end if;
(n ≥ ∀k ≥ j) select(k)=select(j)+k-j+1;; return seq{ti{select}};
end nexnpow;
```

Of course, a complete power set generator is easily defined using this routine.

```
definef nexpow(s,more);
if n more then more=t; n=0; mor=f; return nl;
if mor then return nexnpow(n,s,mor);;
if n eq #s then more=f; return Ω; else
n=n+1;mor=f; return nexnpow(n,s,mor);;
end nexpow;
```

Again using the same idea, we may define a routine which generates all the maps of one set into another.

```
definef nexmap(from,to,more);
if n more than more=t; fseq=nl; (∀x from) fseq(#fseq+1)=x;;
tofol=nf; prev=Ω; (∀xεto) if prev ne Ω then <prev,x> in tofol;
          first=x;; prev=x; end ∀x;
map = {<x,first>, xεfrom}; return map; end if n;
if n (#from ≥ j[j]≥1|tofol(map(fseq(j))) ne Ω) then
          more=f; return Ω; end if;
map(fseq(j)) = tofol(map(fseq(j)));
(j < ∀k ≤ #from) map(fseq(k))=first;; return map; end nexmap;
```

Here is a generator of all the partitions of n,(i.e. distinct combinations of positive integers with sum n), constructed along similar liner.

```
definef nexpart(n,more);
if n more than more=t; pseq={<j,1>, 1≤j≤n};
          return tl(pseq);
if #pseq eq 1 then more=f; return Ω;;
pseq(#pseq-1) = pseq(#pseq-1)+1; k=pseq(#pseq)-1;
pseq(#pseq)=Ω; (1 ≤ ∀j ≤ k) pseq(#pseq+1)=1;; return tl(pseq);
end nexpart;
```

Next we generate some more complex data structures, always using the rather similar technique: however, since trees have a recursive structure, the advance from one tree to another is most readily coded as a recursive process. First consider the problem of generating all *binary trees* with n nodes. We order

these trees by the following recursive principle: suppose that all trees of less than n nodes have already been ordered. Then, among trees with exactly n nodes, we put all those with a smaller number of nodes down the left-hand branch from the root ahead of those with a larger number of nodes down the left hand branch from the root. Among those with a given number of nodes down the left hand branch from the root, we order trees first according to their left hand subtree, then according to their right hand subtree. The first tree is that in which no node has a left hand successor. To advance from a tree to the next tree, we

i. Advance the right-hand subtree to its successor, if possible.

ii. If this is impossible, advance the left-hand subtree, and set the right-hand subtree to the first tree with the same number of nodes.

iii. If this is impossible, transfer one node from right to left, and set both the right and left-hand subtrees equal to the first tree with the appropriate number of nodes.

In SETL, this routine is as follows:

```
definef nextree(n,more);
if n more then more=t; l=nl; r={<j,j+1>,1≤j<n}; top=1;
        return <l,r,top>; end if;
advance (top,more); return if more then <l,r,top> else Ω;
define advance(top,more); nextree external l,r;
if r(top) eq Ω then advance(l(top),more); return;;
advance(r(top),more); if more then return;;
nodsetl=if l(top) eq Ω then r(top)
        else nodesof(l(top) with r(top);
nodesetr = nodesof(r(top)) less r(top);
l(top) = newtree(nodesetl); r(top)=newtree(nodesetr); return;
end advance;
definef nodesof(top); nextree external l,r;
return if top eq Ω then nl else
        nodesof(l(top)) n nodes of (r(top)) with top; end nodesof
```

```
definef newtree(set); nextree external l,r;
        if set eq nl then return Ω;;
prev=Ω; (∀x ∈ set)  l(x)=Ω;
        if prev ne Ω then r(prev)=x;; prev=x;
        end ∀x; r(prev)=Ω;
end newtree;  end newtree;
```

As a final example in this section, we consider the problem of generating all *unordered trees* with n nodes. Such a tree is defined by a set of n nodes, one of which is designated as the head of the tree; and a multi-valued mapping *succs* of this set into itself, which defines the set of successors of each node. We require that the iterated application of this map to the head should eventually produce every other node; and that, for each node x, the iterated application of this map to x can never produce x. This last condition amounts to requiring that a tree be *cycle free*. We assign a standard *sequence number* to unordered trees recursively, as follows. Suppose that sequence numbers have already been defined for all unordered trees of less than n nodes. Given a tree of n nodes, take its root x and take all the subtrees whose roots are the immediate successors of x. Arrange these successors first by decreasing order of the number of nodes they contain, and, if two contain the same number of nodes, then in decreasing order of their sequence numbers; we call this order the standard ordering of the unordered tree. This associates a sequence S of sequence numbers with each unordered tree T of n nodes. By arranging all such sequences S in lexicographic order, and assigning the position of S in this lexicographic order to T its sequence number, we define a sequence number for each unordered tree with n nodes.

In this standard sequencing of trees with n nodes, the first is that in which every root has n-1 successors. To advance from a tree T to the next tree, we

i. Take T in its standard ordering;

ii. Among the subtrees whose roots are immediate descendants of the root R of T, find the last L which it might be possible to advance. This is the last tree which either has fewer nodes than, or the same number of nodes and a lower sequence number than, its immediate predecessor.

iii. If possible, advance L, and make all the nodes belonging to later subtrees immediate successors of R.

iv. If L cannot be advanced, collect all the nodes belonging to later subtrees into a set S. Transfer one node out of S, into L. Redefine L as the first tree with its new number of nodes; make all the nodes remaining in S into immediate successors of R.

v. If the root of L is the last descendant of R, find the last immediate subtree of R prior to L which it might be possible to advance, and proceed with L' as in case iv.

We may write this algorith in SETL as follows.

```
definef nexotree(n,more);
if more then go to try; else
/* initialize */
more=t; top=1; ocesor=nl;
numnodes={<top,n>}; seqno={<top,1>}; hang(top,1,{j, 1<j<n});
/*the subroutine 'hang' adds a set of nodes as immediate
   successors of a given node x, starting at a given position j,
   each node hung is assigned n mmnodes=1;seqno=1; all prior
   immediate successors of x, beginning at the position j,
   are removed*/
cesors=nl; end if;
[ret:] (1 ≤ ∀node ≤ n) cesors(node)=tl(ocesor(node));;
          return <top,cesors>;
[try:] advance (top,more); if more then go to ret; else return Ω;;

define advance (top,more); nexotree external ocesor,numnodes,seqno;
          k=#ocesor(top);
```

```
[look:]  if n  1 < 3[j] ≤ k | nummodes(ocesor(top,j)) lt
         nummodes(ocesor(top,j-1)) or seqno(ocesor(top,j))
         lt seqno(ocesor(top,j-1)) then j=1;;
[adv:]   advance(ocesor(top,j),more);
         set = {u : j < k ≤ #ocesor(top)] nodes(ocesor(top,j));
         if n more than go to movenode; else hang(top,j+1,set);
         return;;
[movenode:]  if set eq nil then go to backup;;
         set1=nodes(ocesor(top,j));
newtop = ∃ set; <nummodes(newtop),seqno(newtop)>=<#set1+1,1>;
ocesor(top,j) = newtop; hang(newtop,1,set1);
  hang(top,j+1,set less newtop);
return;
[backup:]  if j eq 1 then return; else k=j-1; go to look; end if;
end advance;

define hang(top,j,set); nexotree external ocesor,nummodes,seqno;
k=j; (while ocesor(top,k) ne Ω doing k=k+1;) ocesor(top,k)=Ω;;
k=j; (∀nod ε set) ocesor(top,k)=nod;
  <nummodes(nod),seqno(nod)>=<1,1>;
ocesor(nod)=Ω; end ∀nod; end hang;
definef nodes(top);nexotree external ocesor;
return {u: 1 ≤ j ≤ #o esor(top)] nodes(ocesor(top,j))
         with top; end nodes;
end nexotree;
```

6. Various graph-theoretical algorithms related to the optimization of compiler-generated code.

Many interesting algorithms for the optimization of compiler-generated code are based on the analysis of an abstract graph representing the flow structure of the program. This graph, the so-called *program graph* of the program, is defined as follows. Call a sequence of instructions in a program which has the property that control always enters the sequence at its first instruction and always leaves the sequence at its last instruction a *basic block*. If the last instruction in one basic block b_1 might transfer control to the first instruction in another block b_2, call b_2 a *successor block* of b_1. The nodes of a program graph are then the basic blocks of a program, and the successors of a node its successor blocks. The node containing the first (entry) instruction of a program we call the *entry node* e; any node terminating in an "exit" instruction is considered to have no successors, and is called an *exit node*. We suppose that every node of a program graph is reachable from its entry node through a chain of successors; any node not reachable in this way represents code that can never be executed and may as well be deleted from the program. In what follows, any sequence of program-graph nodes in which the n-th is always a successor of the n-1-st will be called a *path*.

Various graph-theoretical notions related to the program graph of a program are useful in analyzing the flow of control and data relationships during program execution. The first of these relationships which we shall consider is that of *backdomination*. A node b is said to backdominate a node c if every path from the entry node to c must pass through b. The backdominators of c are useful in various ways in optimization; for example, under certain conditions, code can be moved from c to certain of its backdominators.

Here is an algorithm, due to F. Allen, for finding all the backdominators of all the nodes in a program graph. First observe that if, starting from the entry node, we can reach (find a path to) a node c without passing through a node x; then we can reach any successor of c without passing through x, except, of course, if x is c. Designate the set of such nodes x as *notneededtoreach*(c). Thus, the set of backdominators of c consists of all the nodes not contained in *notneededtoreach*(c). Here is the SETL algorithm.

```
definef doms(nodes,entry); optimizer external cesor;
/*'nodes' is set of program nodes, 'entry' is entry node,
  'cesor(x)' is set of all successors of node x.
  Returns set 'dom' such that dom(x) is set of all
  backdominators of x */
/* initially, no nodes needed to reach entry */
notneeded to reach={<entry,nodes less entry>}; todo={entry};
(while todo ne nl) node from todo; (∀ c ε cesor(node) |
  (new is{rsnotneededtoreach(node)|n(r eq c or
  r ε notneededtoreach(c))} ne nl)) notneededtoreach(c)
  =notneededtoreach(c) u new; c in todo
/*must process successors of c*/  end ∀c; end while todo;
return{<node,{bsnodes|n bsnotneededtoreach(node)}>,
   node ε nodes less entry} with <entry,nl>; end doms;
```

Next we give an algorithm, due to Earnest, Balke, and Anderson, for linearizing a program graph in an advantageous order. This is, among other remarks that may be made concerning it, an order in which all the backdominators of a node definitely precede the node. The algorithm is as follows:

i. starting at the entry node, and always without repeating any nodes, generate a path.

ii. when this path can no longer be extended, back up along it to a node x from which a new node of the graph is seen as a successor. Starting at this new node, generate

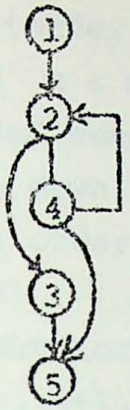
a path; and insert it in linear order into the old path, immediately after x. Continue until the whole graph is linearized.

In SETL, we have:

```
definef graphord(nodes,entry); optimizer external cesor;
order={<1,entry>} /*entry node is first*/;
mark = nl /* mark nodes as processed */
jlast = 1 /*jlast is highest numbered node from which
   new path may begin*/;
(while jlast ≥ ∃[j] ≥ 1, ∃[last]cesor(order(j))|mark(last)ne t)
/*start new path */ path={<1,last>}; mark(last) = t;
/*and extend as far as possible */
(while ∃[next] ε cesor(last)|mark(next) ne t)
   path (#path+1)=next; mark(next)=t; last=next; end while ;;
/*make room for path */
(∀order ≥ ∀i > j) order(i+#path) = order(i);;
/* and insert path after jth node in order */
(1 ≤ ∀i ≤ #path) order(j+i)=path(i);;
jlast=j+#path-1 /*note path(#path) has no unmarked successor;*/;
end while jlast; return order; end graphord;
```

A program graph straightened by the preceding algorithm is said to be in *straight order*. But in this order it may still contain configurations, such as the following,

which it is advantageous to rearrange as



This may be accomplished by applying a *loop-cleansing* procedure, also due to Earnest, Balke, and Anderson, which may be shown to preserve the principal properties of the ordering established by the preceding procedure. The loop-cleansing procedure is as follows.

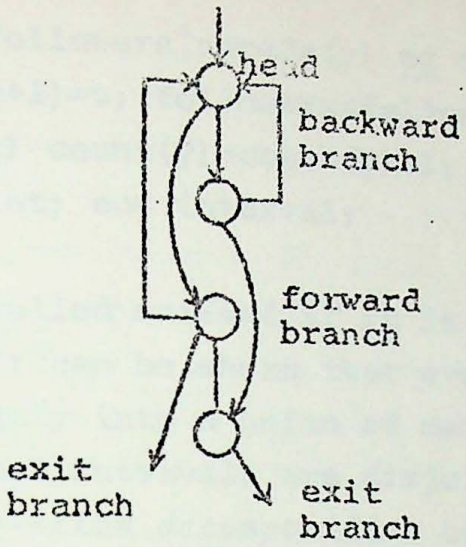
i. Process the nodes n from last to first, in straight order.

ii. Find all nodes following n from which n may be reached by a backward branch. Take these nodes m, and process them in their straightened order, as follows. For each m, mark it, its predecessors, their predecessors, etc.; call the last node marked in this way ℓ. Then push all the marked nodes up toward n and all the unmarked nodes down toward ℓ, keeping the marked nodes however in their established order, and the unmarked nodes in theirs. Continue in this way for each n through all m, and through all n in the reverse of their original straight order.

```
/* generate predecessor map*/
   pred = nℓ; (∀x ε nodes, y ε cesor(x)) x in pred(y);;
/* generate node number */
   number ={<order(i),i>, 1≤i≤#nodes};
 (#nodes ≥ ∀n ≥ 1) head=order(n); latches={nodepred(head)
   |number(nod) gt n}; k=n+1;
/*nodes with indices at least  k  are considered to be 'unmarked'*/
   (while latches ne nℓ) /*find element of minimum index*/
mn=[min: nodelatches] number(nod); m=order(mn);
```

```
marked = closure(psed,{m},{order(j), k ≤ j ≤ #nodes});
latches = {x ε latches | ∩ x ε marked};
/*since nodes once processed need never be processed again */
/* shift unmarked elements down, others up */
(while k ≤ ∃[j] < #nodes|∩ order(j) ε marked
   and order(j+1) ε marked)
<order(j), order(j+1), number(order(j)), number(order(j+1))>
   = <order(j+1), order(j), j+1, j>; end while k;
k = k + #marked; end while latches; end ∀n;
```

Next we turn to study certain notions, useful in the analysis of program graphs, due to J. Cocke and F. Allen. An *interval* in a program graph is a set s of nodes, containing a distinguished node x called the *head* of s, such that there is no entry into s except through x, and such that when x is removed, s is free of loops (a *loop* is a closed path in a program graph). The following figure shows a typical interval:



Single entry regions of this kind serve to make specific the heuristic notion of "inner loop". They are very useful for program flow analysis, and a systematic theory can be built upon them.

It is a characteristic property of intervals that their nodes can be enumerated in such a way that, with the exception of branches terminating at the interval head, all branches between nodes of the interval are "forward" branches, i.e., go from a node x to a node y having a larger serial number in the enumeration of the interval.

The *interval of a node* x is the largest interval with x as head; it may consist of x only.

Here is an algorithm which determines the interval of a node x, and enumerates the nodes of this set in an order having the property noted above. It also finds the set of all successors of nodes of the interval which do not belong to the interval; this will be needed below.

```
/*before entry to interval finding routine, count the number
  of predecessors of every node*/ npreds={<x,0>, xenodes};
(∀xenodes, ycesor(x)) npreds(y)=npreds(y)+1;;
definef interval(x); optimizer external cesor, npreds, followers;
int=nl; followers={x}; count ={<y,0>, yenodes};
/*'count' will be a count of the number of predecessors of a
  node which belong to the interval being constructed*/
```

```
(while(newin is{y∈followers|npreds(y) eq count(y)}) ne nl )
(∀z∈newin) int(#int+1)=z; followers=followers less z;
(∀y∈cesor(z)|y ne x) count(y)=count(y)+1; y in followers;; end ∀z;
end while; return int; end interval;
```

An interval is called *maximal* if it is not contained in any larger interval. It can be shown that every program graph can be decomposed uniquely into a union of maximal intervals, and that distinct maximal intervals are disjoint. This decomposition, the so-called *Cocke-Allen decomposition* of a program graph, is quite useful in flow analysis.

To find these maximal intervals, we proceed as follows. Take the interval generated by the entry node of the program graph; this is a first maximal interval. Then, iteratively having formed other maximal intervals, take any successor x of a point in these intervals not lying in any of them, and form the interval of x; this is a new maximal interval.

The following SETL algorithm realizes this process. It also associates, with each maximal interval, the set *follow(int)* of all nodes which are successors of a node of the interval without belonging to the interval; and with each node b of the program graph the maximal interval intov(b) which contains it.

```
definef intervals(nodes,entry); optimizer external cesor,
   followers, follow, intov; ints=nl; seen={entry};
   (while seen ne nl) node from seen; i is interval(node) in ints;
follow(i)= followers; (∀b∈ tl[i] /*tl[i] is the set of all
   nodes in interval */) intov(b) = i;; seen=seen u followers;
end while; return ints; end intervals;
```

The *derived graph* g' of a program graph g is defined as follows: the nodes of g' are the intervals of g; the successors of an interval *int* are the intervals distinct from *int* containing successors of the nodes within *int*; the entry node of g' is the interval containing the entry node of g. The derived graph of a program graph gives a simpler, "coarsened" representation

of its program flow, in which representation "inner loops", i.e., intervals, appear as single points, and only "outer" loops are shown. If g contains any interval of more than one point, g' will have fewer nodes than g. A program graph in which no interval of more than one point can be found is called an *irreducible graph*; fortunately, such graphs arise only rarely in connection with actual programs. In SETL, we may write the criterion of irreducibility very simply as follows:

(#nodes) eq #intervals(nodes,entry) .

The definition of the derived graph in SETL is also quite trivial; here it is:

```
definef dg(nodes,entry); optimizer external cesor,
        follow, intov, dent;
ints = intervals(nodes,entry); dent = intov(entry);
(∀i ε ints) cesor(i)=intov[follow(i)];; return ints; end dg;
```

By forming successive derived graphs g', (g')', etc. of an original graph g, we look at an original program in a more and more global way. In cases in which this *derivation sequence* converges to a graph consisting only of a single node, we may claim that the method of intervals provides a decisive analysis of program flow. Graphs having this property are called *eventually reducible* program graphs.

Here is the SETL definition of the derived sequence of a graph.

```
definef dseq(nodes,entry); optimizer external dent;
seq={<1,nodes,entry>}; <n,e>=<nodes,entry>;
(while # der is $ dg (n,e)  lt  n doing <n,e>=<der,dent>;)
   seq(#seq+1) = <der,dent>;; end dseq;
```

Intervals have many uses in the optimization-analysis of programs. Generally speaking, the derivation sequence of a program gives a very useful guide to the order in which various optimizing processes can most effectively be applied. We shall now discuss one such application, studying the so-called "live-dead" analysis of variables.

A variable x is said to be *live* at a given point q in a program if from this point there exists a path P in the program, not passing through any instruction which assigns a value to x, and such that P terminates in an instruction which uses the value of x. In the contrary case, x is said to be *dead* at q. It is clear that if x is live at given point in a program, its value must be saved in some known register or core location for later use. Conversely, when x becomes dead, the register containing it becomes available for other use.

Next we intend to describe an algorithm, due to K. Kennedy, for efficiently deriving live-dead information for any eventually reducible program graph. Of course, to carry out such an analysis, we require that basic information concerning the use of variables within a program be available. We shall suppose that this information is made available according to the following conventions.

i. All the variables with which the program is concerned are collected into a set of *vars*;

ii. With each basic block b is associated a set uses(b), consisting of all the variables x in the set *vars* such that, starting at the entrance to b, a use of x will be reached before any assignment is made to x;

iii. With each basic block b is associated a set thru(b), consisting of all the variables x in the set *vars* such that, starting at the entrance to b, the exit of b will be reached before any assignment is made to x.

A path through the program which does not encounter any instruction assigning a value to a variable x is called an x-*clear* path. We shall now wish to associate certain sets with each interval *intv*. These are as follows:

1. The set of all x such that there is an x-clear path in *intv*, starting at the entrance to *intv* (i.e., immediately before any instruction in *intv* is executed) and terminating at a use of x within *intv*.

2. For each block e of *follow(intv)*, the set of all x such that there is an x-clear path through *intv*, starting at its entrance, and terminating at e.

We shall call the first of these sets *uses(intv)* and the second *thru(intv,e)*. The following observation allows these sets to be calculated easily. If b is the head of *intv*, and if the blocks of *intv* are enumerated in the order defined by the basic interval construction function *interval(x)* described above, then, as has been remarked, all backward branches in *intv* go through b. Thus all irredundant paths from the entrance to b i.e., paths through *intv* not containing pointless repetitions, will consist of forward branches only. Therefore the information we require can be developed by a simple scheme of propagating uses backwards, considering forward branches only. In SETL, the algorithm we require is as follows:

```
naux=nl; taux=nl; head=intv(1);
(#intv ≥ ∀n ≥ 1) b=intv(n); forward={yεcesor(b)|
   intov(b) eq intv and y ne head};
naux(b)=uses(b) u thru(b) int$ [u : y ε forward] naux(y);
(∀intx  ε cesor(intv))if intx(1)εcesor(b)
   then taux(b,intx) = thru(b);
else taux(b,intx) = thru(b) int[u: yεforward] taux(y,intx);
end else; end ∀ intx; end ∀n;
uses(intv)=naux(head); (∀intxεcesor(intv))
   thru(intv,intx)=taux(head,intx);;
```

We may now observe that uses(intv) and thru(intv,inta) are related to the interval intv in just the same way that uses(b) and thru(b) are related to a basic block b. Thus, the construction shown above can be repeated for the derived graph, and therefore, iterating, for all the graphs of the derivation sequence. We write the process to that builds this information for all the graphs of the derivation

sequence as a subroutine; except for the slight differences occasioned by the fact that intervals may have several exits while a basic block can have only one, we may use almost precisely the code which appears above. With the necessary corrections, we have:

```
define builda(nodes,entry); optimizer external cesor,
  intv, uses, thru, seqd;
seqd = dseq(nodes,entry); (1 < ∀k ≤ #seqd, intv ε hd seq(k))
naux=nl; taux=nl; head=intv(1); (#intv ≥ ∀n ≥ 1) b=intv(n);
orward ={yεcesor(b) |intov(b) eq intv and y ne head};
f k eq 2 then naux(b)=uses(b) u (thru(b) int[u:yεforward]naux(y));;
(intxεcesor(intv)) if intx(1) ε cesor(b)
 then  taux(b,int x) = thru(b);
ele taux (b,int) = thru(b) int [u: yεforward] taux(b,intx);
en else; end ∀intx ; else /*k gt 2 */
naux(b) = uses(b) u [u : y ε forward] (thru(b,y) int naux(b,y));
(∀intx  ε cesor(intv)) taux(b,intx)=(if intx(1)εcesor(b)
  hen thru(b,intx(1)) else nl)
[u: y ε forward](thru(b,y) int taux(y)); end ∀intx; end else;end ∀n;
uses(intv)= naux(head); (∀intxεcesor(intv)) thru(intv,intx)=
  taux(head,intx);;
end ∀k; return; end builda;;
```

Next we may make the following observations. Since when the blocks b of an interval *intv* are enumerated in the order we have been considering, every backward branch passes through the head of *intv*, it follows that an irredundant path from b thru *intv*, to either the use of a variable or to an exit from *intv*, will consist either of forward branches exclusively or will contain one and only one backward branch. Next, suppose that we are dealing with a program graph that is eventually reducible. Then the graph g_{n-1} appearing at the next-to-last stage in the derivation consists of a single interval *without any exits*; hence the observation that we have just made gives an easy way of computing uses(b) for each block b

of g_{n-1}. But the blocks of g_{n-1} correspond exactly to those blocks of g_{n-2} which are interval heads; hence, working backwards iteratively, we can compute uses(b) for each block of $g_{n-2}, g_{n-3}, \ldots$, until g is reached. We will then know, for each basic block b in our original program, whether or not there exists an x-clear path from the entrance to b reaching a use of x; and this evidently tells us whether or not x is live at the entrance to x.

We may write the routine which completes the construction of *uses* in SETL as follows:

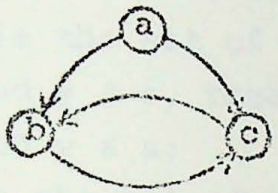
```
define builduse(nodes,entry); optimizer external cesor,
   intov, uses, thru, seqd;
builda(nodes,entry); (#seqd ≥ ∀k > 1, intvehd seq(k))
(#intv ≥ ∀n ≥ 1) b=intv(n); backorexit={cecesor(b)|
   n ceintv or c eq intv(1)};
uses(b) = uses(b) u
   (if k eq 2 then
thru(b) int [u : c ε backorexit] uses(intov(c));
else
[u: cεbackorexit](thru(b,c) int uses(intov(c)));
end ∀n; end ∀k; return; end builduse;
```

Note that as it stands this algorithm is valid only for program graphs which are eventually reducible. It is also worth observing that the set

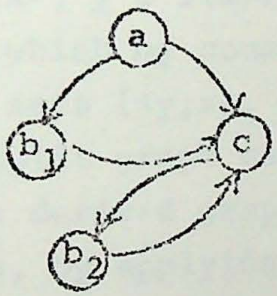
uses(entry)

i.e., the set of variables which are live immediately on entry to the program, is in fact the set of improperly initialized program variables, concerning which we would, in a total optimizing-compiler system, wish to issue a diagnostic message.

If no irreducible graphs (consisting of more than one point) are encountered, the derivation sequence of an originally given program graph g will have the kind of use which the last few algorithms have illustrated. Irreducible graphs are obstacles to the application of such algorithms, fortunately, there are methods for dealing even with them. To see what these might be, note that the simplest example of an irreducible graph is as follows:



If for the moment we think of the code represented by the node b as being duplicated into two copies, one of which, b_1, is entered from a, while the other, b_2, is entered from c, then we see that a perfectly equivalent program which has the following flow can be found:



In this graph $\{a,b_1\}$ and $\{c,b_2\}$ are intervals; so its derived graph consists only of two nodes, and the derived graph of this consists only of one. This example shows that an abstract process of "node splitting" can allow the reduction even of graphs which originally are irreducible. Moreover, the phenomenon to be observed in this example is perfectly general. To see that this is the case, we argue as follows. Let g be a program graph consisting of n nodes. Let s be a set of nodes within g, including the entry node of g, consisting of as few nodes as possible, and such that when s is removed, the remaining set $\bar{s}$ of nodes admits no loops; this is to say that at least one point of every loop belongs to s. Note that

since every loop contains at least 2 nodes, s will never have more than n-1 points. For each x ε s, let f(x) be the set of all points in $\bar{s}$ reachable along a path starting at x all of whose nodes but x lie in $\bar{s}$. We then construct a *split graph*, as follows. The points of this new graph are the pairs <x,nℓ> for xεs, together with the points <y,x> where y ε f(x), these points represent multiple "split copies" of y. In this collection, we define the successor relationship as follows:

i. cesor(<x,nℓ>) is the set of points <y,x> for which y ε cesor(x) and y ε $\bar{s}$, together with the set of points <y,nℓ> for which y ε s;

ii. cesor(<y,x>) is the set of points <z,x> for which z ε cesor(y) and z ε $\bar{s}$, together with the set of points <z,nℓ> for which z ε cesor(y) and z ε s.

It is clear from this definition that, in the split graph, each of the points <y,x>, y ε f(x), can be reached from <x,nℓ>; moreover, the set {<y,x>, y ε f(x)} contains no cycles, since if it did so would $\bar{s}$, which by construction we have ruled out. Therefore each of the sets {<y,x>, y ε f(x)} with x is an interval. Hence our split graph can be covered by at most n-1 intervals, and its derived graph will therefore contain at most n-1 points. Thus, by applying the node splitting process whenever an irreducible graph is encountered, we can construct a generalized derivation sequence which will always converge to a graph consisting of only one single node.

To find, within a graph g, a set s of nodes of the kind required, we may proceed as follows. Arrange the nodes of g in sequence, using the ordering algorithm described in the first part of the present section. Then define s as follows: first put all backdominators into s. Then take all backward branches whose origin node u does not belong to s; suppose that the target node of the branch is v. If every backdominator of u is a backdominator of v, put v in s. In SETL, the algorithm and the remaining parts of the constructon of the split graph appear as follows.

```
definef graphsplit(nodes,entry); optimizer external newat,cesor
seq=graphord(nodes,entry); domsof=doms(nodes,entry);s=tl{domsof};
seqno={<seq(j),j>, 1 ≤ j ≤ #seq};
(#seq ≥ ∀j > 2|n seq(j)εs) a=seq(y);
(∀bεcesors(a)|seqno(b) lt j) if n domsof{a} incs domsof{b}
  then b in s;;
end ∀b; end ∀j; sbar={bεnodes|n bεs};
newg ={ <x,nl>, xεs} u {<y,x>, xεsbar, yεclosure(cesor,{x},sbar)};
(∀xεs) cesor(x) ={<y, if y ε sbar then x else nl>, yεcesor(x)};
(∀y ε closure(cesor,{x}, sbar)) cesor(<y,x>)
  = {<z, if zεsbar then x else nl>, zεcesor(y)};
end ∀y; end ∀x;
newat = <entry, nl>; return newg; end graphsplit;
```

With regard to the use of a split graph for the type of analyses in which we will be interested, as for example in the derivation of live-dead information, we may make the following remarks. In the generalized version of the routine *builda*, each of the nodes <y,x> derived from y by splitting will inherit the set *uses* of variables used within it from y, i.e., we will have uses(<y,x>) = uses(y). Similarly, we will have thru(<y,x>,<$\bar{y}$,x>) = thru(y,$\bar{y}$) if y has the successor $\bar{y}$ in $\bar{s}$, thru(<y,x>,<$\bar{y}$,nl>) = thru(y,$\bar{y}$) if y has the successor $\bar{y}$ in s, etc. Then, in builduse, we will derive use(y) for a node that must be split into nodes $y_1,\ldots,y_n$ as the union of use(y_1),...,use(y_n). We leave it to the reader to write out these algorithms in their generalized form.

Packing algorithms are used in optimizers to assign quantities to registers in an effective way. We shall conclude this section by giving two algorithms of this kind. In each case, we assume that the quantities to be assigned form a set s on which a map *inter* is defined; if x ε inter(y), then x and y are "interfering" quantities and cannot be assigned to the same register. The algorithms to be described will establish a map *rep* on s; each

y ε s will then be assigned the same register as rep(y), so that only those x with rep(x) eq Ω need be assigned registers to begin with, and the other assignments will follow automatically. The number of registers required is then equal to #{xεs|rep(x) eq Ω}; we wish to reduce this quantity to a minimum. Naturally, if rep(x) and rep(y) are to be the same, x and y must be non-interfacing.

The first algorithm, due to J. Cocke, begins for arranging the quantities x in decreasing order of #inter(x). The first quantity x in squence is then removed, and the remaining quantities scanned in sequence to find quantities y for which rep(y) may be assigned as x without violating the interference condition. Whenever rep(y) is defined, y is removed from the sequence. This process repeats as often as necessary, as shown in the following SETL algorithm, wherein we presume that the sequence seq of quantities x arranged in decreasing order of #inter(x) has been pre-calculated.

```
rep=nl; marked=nl; low=0; (while low< ∃[j] ≤ #seq|
  n j ε marked) j in marked; x = seq(j);
interfere=inter(x); k=j; low=j;
(while k < ∃[kk] ≤ #seq|n y is$ seq(kk)εinterfere
  and n kk ε marked  doing k=kk;)
kk in marked; interfere=interfere u inter(y);
  rep(y) = x; end while k;
end while low;
```

Suppose in the above algorithm that #inter(seq(j)) = n_j. It is clear that, if j is not marked before the k-th iteration of the outer while-loop in the above algorithm, then seq(j) interferes with at least k quantities preceding it in sequence; hence $k \leq \min(j-1, n_j)$. The maximum possible value of k, which is plainly also the maximum number of "registers" needed for the packing by the above algorithm of all quantities, is therefore at most

(1) $\max_j \min (j, n_j+1)$.

If the total number $n = \sum n_j$ of interferences is prescribed, the quantity (1) will clearly not decrease if whenever $n_j+1 < j$ we increment n_{j+1} and decrement n_j, provided that initially $n_{j+1} < n_j$. Similarly, if $n_j+1 > j$ we may decrement n_j to increment n_{j-1}. This worst case consideration makes it plain that for given n (1) will attain its maximum approxiately when for some j we have $n_1 = \ldots = n_j = j-1$ and $n_{j+1} = n_{j+2} = \ldots = 0$; and then clearly j is the number of "registers" needed for the packing of all quantities. Since then $n = j(j-1)$, this shows that if a total of n inter-quantity interferences exist, approximately $\sqrt{n}$ "registers" should suffice to contain all quantities.

This line of thought suggests an improved packing algorithm, proposed originally by A. P. Ershov. If we examine our preceding packing algorithm, we see that when rep(y) is assigned as x, the total "effective" set of interferences becomes inter(x) u inter(y) . If we then choose y so that #(inter(x) int inter(y)) is as large as possible, the largest possible number of the interferences originally present will become irrelevant. A SETL algorithm expressing this procedure is as follows:

```
/*quants is the total collection of quantities being considered;
curq those to which no representations have yet been assigned;
curint the current state of interferences, taking the
assignment of representatives into account; we write
y ε repdby(x)  if, as far as our calculation yet shows,
Y is to be represented by x */
curq = quants; curint=inter; repdby={<x,{x}>, xεquants};
/* determine pair to identify, if any */
```

```
[loop:]  intno=0;
(∀x∈curq, y∈curint[curint(x) int curq] int curq|n y∈curint(x))
if (n is #(curint(x) int curq int curint(y)}) gt intno
   then  /*better found*/
intno=n; best = <x,y>;; end ∀x;
if intno eq 0 then go to done;
/*else identify y with x */
<x,y> = best; repdby(x)=repdby(x) u repdby(y);
curint(x) = curint(x) u curint(y);
(∀z ∈ curint(y) int curq) curint(z)=curint(z) with x;;
curq=curq less y;
go to loop;
[done:]  rep={<y,x>, x∈curq, y∈repdby(x)|y ne x}
```

7. Parsing algorithms.

In the present section we will describe a set of general algorithms which serve to define efficient table-driven parsers for a wide class of programming languages. This class of parsers will also generate diagnostic messages in a useful standard form when defective programs are encountered. We shall then use the scheme which is to be developed to specify, in full detail, the parser for a simple algebraic language somewhat more general than the Dartmouth BASIC language.

Generally speaking, parsing constitutes (after a lexical scan) the first stage of a total compilation process. A lexical scanner breaks an input stream of characters into a string of *tokens*. The parser then associates a tree, called the *parse tree*, with this string of tokens. This tree is valuable for the following reason: it establishes a fixed system of *tree addresses* for the tokens of the sentence parsed, and does this in such a way that *tokens playing given roles within a sentence will always be found at fixed corresponding tree addresses*. For example, in statements, statement-type determining keywords will always be found at certain particular tree nodes; in expressions, the *central operator sign*, i.e., that sign denoting the operation to be performed last, will always be found at certain particular tree nodes, etc. This is of course not the case for the string of tokens originally representing a sentence.

Our parsing process will be divided into two stages, which we call *preparse* and *postparse* respectively. During preparse, we apply a simple high-speed precedence method to assign a tree to each input string of tokens; we require of this tree only that it be as correct as possible in as many cases as can be managed by appropriate choice of precedences. During postparse, we check the structure of this tree, correcting preparse-assigned provisional structures as necessary, and issuing diagnostics when true errors are encountered.

The preparse will have the following structure. A routine like the *nextoken* routine described in section 4 will be used for lexical analysis. A set of reserved words will be specified; moreover, every token which can be received from *nextoken* will be classified as belonging to one of the five following gross types:

i. Name type; this will also include constants of various kinds. Designation: 'n' or 'var'

ii. Operator type; this will also include most reserved keywords. Designation: 'o'.

iii, iv. Left and right parenthesis type; These types may include parentheses of various kinds, including keywords which serve as parentheses, such as the *begin* and *end* of ALGOL. Designations: '(' and ')'.

v. Sentence delimiter. This consists of whatever unique symbol (possibly an end-record mark) is used to terminate sentences of the language to be parsed. We use the same mark as a normal beginning-of-sentence sign in our preparse description. Designation: Λ or er.

The preparser is not to be responsible for any error detection; it will parse correctly formed sentences in a useful way, but will assign *some* parse tree to any arbitrary string of tokens. Its action may however depend on numerical precedences separately assigned to particular operators, and to non-reserved tokens by their lexical types; we also allow separate left and right procedures. Note that in this very general style of precedence parsing, we assign left and right precedences to every symbol, including variable names and constants. The precedences assigned to such names may be thought of as precedences belonging to elided "catenation" operators. Given this strategy, we will want to be sure that the preparser is capable of handling every situation, however bizarre, which can arise. In particular, we will need rules which can cover cases in which two successive names or numbers occur in immediate sequence without any operator separating them, a situation which might arise for certain

statements in certain languages. We will also need rules to cover bizarre cases such as))(, etc. Moreover, rules which cover all the special situations which can arise at the beginning and end of statements will be required. Arguing then by induction on the length of the string to be treated, it is not hard to see that the following set of rules will allow a precedence parser to condense any sequence of symbols whatsoever.

Context Up to Last Symbol Scanned	Action
o n	Compare precedence p of o to precedence p' of next following symbol. If p exceeds p', condense two symbols if context is oon, (on, or Λon, but three symbols if context is n o n .
n n	Compare precedence p of first n to that of next following symbol; condense two symbols if p is greater.
(n)	Condense 3 symbols unless left parenthesis is preceded by an n, in which case condense 4 symbols.
(nΛ	Supply specially flagged "missing parenthesis" and proceed as in preceding case.
ΛnΛ	Condensation complete; return from preparser.
( )	Condense, and treat as special null quantity.
o)	Treat as monadic operator requiring no argument.
oΛ	
Λ)	Flag parenthesis, and treat it as a monadic operator.
(Λ	Flag parenthesis, and treat it as a monadic operator requiring no argument.
ΛΛ	Blank statement or illegal endfile condition.

The operation spoken of as 'condensation' in the above table is to be handled as follows. When a condensation is performed, a new tree node is created, whose descendant subnodes become those of the elements condensed which are of type n; elements of other kinds belonging to a condensed group determine an attribute of the new node called its "node type". When a new node is formed, we also regard it as a token of type 'variable', and treat this token as the next input to the preparser; in this way, successive condensations will generate a tree structure. An exception is made to the above rule when condensations are to be progressively applied either to strings nnnnn... or to strings nononono... in which a single operator token is repeated. In this case, we prefer to generate, for the whole string, a single node to the branches of which all the n's in the string are attached; the node type of such a multi-branch node will either be defined by the operator o, or will be 'catenation'. This variation of tree structure is advantageous for various subsequent purposes.

Note concerning the above that even those of the above cases which initially appear bizarre may turn out to be useful. For example, in one or another programming language one might wish to permit a reserved keyword *random*, whose value was to be supplied by a random number generator, and which could be used in such combinations as

$$\Lambda\ a = a + \text{random}\ \Lambda\ .$$

Given our convention that reserved keywords are to be treated as operators, this involves an instance of the initially strange-looking case oΛ noted in the preceding table.

Before proceeding to give a SETL algorithm for the general precedence parse, we wish to note certain simple generalizations of the strategy outlined above, one of which we will actually incorporate in the SETL algorithm. The technique which has been outlined will work nicely whenever all the statements of a language employ approximately the same precedence rules. This, however, is not always the case. For example, within a FORTRAN expression like

F(A/B, C/D) ,

the division sign takes precedence over the comma. On the other hand, in a FORTRAN COMMON statement like

COMMON /A/B, C, D/E/F, G, H

the slash takes precedence over the comma. Most such situations can be covered by the following simple generalization of the technique described above. Let the preparser detect certain special sequences of a few tokens, such as

ΛCOMMON /

When such a configuration is detected, the precedence values of a few special operators can be modified, and parsing can proceed as before. When after return to a main program the parser is re-entered, these precedences must be repaired. Another useful generalization could be as follows. The preparser can test for special situations whenever it treats an (n) case and to modify its precedence tables when certain situations are detected in these parenthesis-balanced situations. This would allow special precedences to be employed within parentheses and then abandoned, a procedure that, e.g., might be useful in dealing with the FORTRAN

FOPMAT(...

The generalized precedence parse routine to be described shortly will incorporate the first, but not the second, of these generalizations. We will aim to write a SETL routine permitting an extremely efficient machine-level realization. To detect the various configurations, shown in the table given above, in which actions of condensation may be called for, we will use a programmed finite state automaton. An automaton of just the type called for can be constructed in a manner originally suggested by Domolki; because of the very elegant way in which Domolki's construction exploits the bit-parallelism normally available in machine-level operations, we shall, in presenting it, allow ourselves to become involved in a machine-related discussion to a degree that we should normally prefer to avoid.

Domolki's device is as follows. Enumerate all the configurations that the finite-state automaton is to detect. In the case we particularly wish to consider, these are

i. o n ii. n n iii. (n)

iv. anb , where a is either '(' or Λ , and where b is either ')' or Λ .

v. cd, where c is either '(', o, or Λ, and where d is either')' or Λ .

vi. xyz, representing some special sequence of three symbols which we wish to detect in order to perform some user-defined special action which it may imply.

We now set up a bit-string, divided into zones each of which corresponds to one of the configurations we wish to detect; the length (in bits) of each zone will be equal to the number of symbols in the configuration to which it corresponds. In the specific case we consider, we will have zones of 2,2,3,3,2,3 bits, and thus a bit-string 15 bits in total length. The states of our finite-state automaton correspond to all the possible values of such a bit-string; note that we therefore deal with an automaton whose approximately 32,000 states we might prefer not to have to enumerate in any much more explicit fashion.

The significance of these bits is to be as follows: the i-th bit (counting from left to right) of the j-th zone is to be 1 if the last i symbols scanned are the first i symbols of the j-th configuration which we wish to detect; otherwise the i-th bit of the j-th zone is zero. Each time a new token t is scanned, the values of these bits will change in rather simple fashion. Writing b(i,j) for the 'old' value of the i-th bit of the j-th zone, and writing $\bar{b}$ for the corresponding 'new' values, we plainly have

i. for i = 1, $\bar{b}(1,j) = 1$ if t is the first symbol of the j-th configuration;

ii. for i > 1, $\bar{b}(i,j) = b(i-1,j)$ if t is the i-th symbol of the j-th configuration, $\bar{b}(i,j) = 0$ if not.

Suppose then that the following bit-strings are available.

a. A bit-string *starts* marking those bits which correspond to the first symbols of any configuration.

b. A bit-string *finish* marking those bits which correspond to the last symbol of any configuration,

c. for each token t, a bit-string m_t marking the i-th position in a zone if and only if t can be the i-th symbol of the corresponding configuration,

then to calculate $\bar{b}$ from b, we have only to apply the following formula:

$$\bar{b} = ((\underline{\text{right}}\ b)\ \underline{\text{or}}\ \text{starts})\ \underline{\text{and}}\ m_t$$

where *right* designates a one-bit right shift. Moreover, some configuration in which we are interested will have been detected if and only if the bit-string b comes into a state in which we have

$$b\ \underline{\text{and}}\ \text{finish} \neq 0\ ;$$

in such a case, the particular bit of b and finish which is nonzero will determine the configuration which has been found.

The SETL algorithm given below embodies the approach to precedence preparsing which has just been described. The following comments will illuminate salient details of this algorithm. As is standard for precedence parsers, the algorithm uses a stack, called *statstak* in what follows, which holds still uncondensed tokens, and in our case also information on the 'kinds' of these tokens, and on the state of the auxiliary finite-state automaton immediately after a token was read. An auxiliary stack (which will never contain more than two items) called *bakstak* contains tokens read as part of any necessary look-ahead but still to be processed. An initialization block called *setup* establishes all necessary tables, analyzing user-supplied information where necessary for the definition of these tables. In particular, information defining special reserved tokens and significant lexical classes may be supplied; tokens having no special significance are classified as belonging to the catch-all category 'variable'.

of course, precedence information covering every possible token-type will be available. In addition, a set of masks defining the effect of every possible token type on the auxiliary finite state automaton which the preparser uses will be available. A 'gross classification function' which assigns every kind of symbol to one of the five principal classes var, o, '(', ')', and er is also used.

In the interests of simplicity we omit all diagnostic checks on the reasonableness of user-supplied information; the initialization routine given in section 4 shows how such checks might be programmed.

The 'nn' and 'non' condensations check to see if a sequence, in one case a sequence of catenations, in another a sequence of identical operators, is to be continued; if this is the case, one enlarges the list of descendants of an old tree node rather than creating a new tree node. With regard to some of the details of the initialization procedure, the following should be remembered: The sequence of bit-groups and their significance, which of course determines the masks required, is taken (cf. i.-vi. above) as

nn|on|(n)|(,er) n (),er}|(,o,er) (),er}|xyz,

where the three last positions xyz are reserved for user-defined special situations. Note in particular that completion of the fourth pattern will always be detected when completion of the pattern (n) is detected. Our initialization allows for this, and also for the fact that treatment of 'nn' and 'on' cases begins together, with separation following.

Because of the number of special situations which must be represented, and because it incorporates an initialization procedure, the algorithm which now follows is reasonably long.

```
define preparse(treetop);
initially do setup;; /*initialization block found below*/
do restart; /*user-supplied block which may be necessary to
        restore precedence table at start of new sentence*/
/*initialize with sentence delimiter*/
statstak=nl; bakstak=[<1,er>]; zero=00000o; state=zero;
        go to jumpin;
/*stack routines used below*/
definef top stk; return stk(#stk); end top;
definef topoff stk; x=stk(#stk); stk(#stk)=Ω; return x; end topoff;
define e onto stk; stk(#stk+1)=e; return; end onto;
definef n elm stk; return stk(#stk-n+1); end elt;
/*begin of main process*/
getoken:<state,kind,tokdat> onto statstak;
jumpin:  if bakstak ne nl then <kind,tokdat>= topoff bakstak;
        else tokdat=nextoken; <type,token,->=tokdat;
        kind=getkind(tokdat); end else;
definef getkind(tokdat); /*auxiliary routine to classify token*/
preparse external symbkind,typkind; <type,token,->=tokdat;
return if (x is symbkind(token)) ne Ω then x else
        if (x is typkind(type)) ne Ω then x else 'var';
end getkind; /*now calculate new state-defining bitvector*/
newstate: state=lb+(len state-1 first state) or starts
        and mask(kind); go to label (state and finish);
/* returns to getoken in case no condensation need be tried*/
nnon: /*name preceded by name or operator*/
        if bakstak eq nl then <getkind(x is nextoken),x>
        onto bakstak;;  <kind2,->=top bakstak;
        <-,kind1,-> =top statstak;
/*note that the elements on statstak have the form
 <state,token kind, token lexical type (if any), token,
 token-associated data (if any)>*/
/*part-finders for statstak*/
definef knd stelt; return <-*->stelt; end knd;
definef tokof stelt; return <*z4>stelt; end tokof;
definef tdat stelt; return <*zt3>stelt; end tdat;
```

```
if rprec(kind2) gt lprec(kind1) then go to getoken;;
/* else must do condensation*/
/*separate nn case from o n case */
if(state and nnmask)ne zero then do condensenn; else
/*separate 'non' from 'on'*/
if gross(knd 2 elm statstak) eq 'var' then do condensenon;
  else new=newat; do condenseon;;
newcycle:  tokdat= new;
newcyc:    kind='var';
restart:   state=hd top statstak; go to newstate;
/* here follow three condensation procedures*/
block condenseon; desc(new,1)=tokdat;
nodtype(new)  = tokof topoff statstak; end condenseon;
block condensenn; /*decide if additional catenation necessary*/
oldel=tdat topoff statstak; if nodtype(oldel)
  ne 'catenation' then new=newat; desc(new,1)=oldel;
  desc(new,2)=tokdat; nodtype(new)='catenation';
  else /*add new element to prior catenated list*/
  desc(oldel,  #desc(oldel) +1)=tokdat;
  new=oldel;  end else; end condensenn;
block condensenon; /*decide if continuation of same operator*/
optok=tokof topoff statstak; oldel=tdat topoff statstak;
if nodtype(oldel)ne optok then new=newat;
desc(new,1)=oldel;desc(new,2)=tokdat;nodtype(new)=optok; else
/*add new element to prior list */
desc(oldel, #desc(oldel) +1)=tokdat;
new=oldel;  end else; end condensenon;
parcas:/* enter here on cases '(n)'; first separate n(n) from o(n)case
new=newat, if gross(knd 3(elm statstak))eq 'var'
  then do condense4(token); else do condense3(token);;
  go to newcycle;
```

```
/*two additional condensation procedures*/
block condense3(t), desc(new,1) = tdat topoff statstak;
nodtype(new)=tokof topoff statstak+t; /*so that the 'type'
of the node will be some concatenated pair of parentheses,
such as '( )'*/
end condense3;
block condense4(t), desc(new,2) = tdat topoff statstak;
nodtype(new)=tokof topoff statstak+t;desc(new,1)
  = tdat topoff statstak; end condense4;
mispar: /*case of missing left or right parentheses;
separate the two cases */
new=newat; if kind ne er then desc(new,1)=tdat topoff statstak;
nodtype(new)='missing (,' + token;
else /*separate n(n from o(n cases*/
if gross(knd(3 elm statstak))eq 'var' then do condense4(',missing)')
  else do condense3(',missing)');; end else;
  go to newcycle;
/*here begins code to deal with various two-token special cases*/
specialcases: kind1=knd top statstak;
go to {<er,')',erp>,<'(',er,per>,<'(',')',pp>,
<o,er,oer>,<o,')',oer>,<er,er,erer>}(gross(kind1),gross(kind));
erp: /*treat left parenthesis as special operator*/
kind = 'o'; go to restart;
per: <kind,tokdat> onto bakstak; tokdat=tdat topoff statstak;
   <type,token,->=tokdat; kind='o'; goto restart;
pp: /*treat as special null quantity*/
   tokdat=<'var',tokof topoff statstak+token>; go to newcyc;
oer: /*supply missing variable to terminal operator */
  <kind,tokdat> onto bakstak; tokdat=<'var','omitted'>;
  go to newcyc;
erer: /*test for blank statement or endfile*/
if n tokdat ε endfilesigns then /*signal blank statement*/
  treetop=<'var','blankstatement'>; return; else
  print 'parse terminated by illegal end of file'; exit;;
users: do usercode; /*user-supplied block to deal with
  user-established special cases*/
```

```
/*here follows the principal initialization block*/
block setup; <var,o,er,on,beg?=;
/*first initialize the bit-vectors 'starts', 'finish',
  and 'nnmask', and the switch 'label(x)' using
  a position-defining 15-tuple*/
begend=<beg,nnon,beg,on,beg,f,parcas,beg,f,mispar,beg,
  specialcases,beg,f,users>;
starts = beg locsin begend;
finish = labs is{nnon,on,parcas,mispar,specialcases,users}
  locsin begend;
label ={< x locsin begend,x>, xεlabs|n xε{on,mispar}};
label(on locsin begend)=nnon; label({mispar,parcas}
  locsin begend) = mispar;
label(00000o) = getoken; nnmask = on locsin begend;
/*the user must declare certain information, to be used next.
  the first pieces analyzed define a 'kind' for each reserved
  symbol and significant lexical type, by giving a set of
  2 and 3-tuples, each of which is either of the form
  <{token set}, kind, gross kind(if kind is not
  var, o, '(', ')', or er)>
  or, for a single token <token,kind,etc.> */
kindz=nl; gross={<x,x>,xε{var,o,'(',')',er}}; do give;
/*    {tokinf,typinf,&prinf,rprinf,triple,endfilesigns)
  user-supplied*/
symbkind=analyze(tokinf); typkind=analyze(typinf);
definef analyze(inf); preparse external gross,kinds; map=nl;
(∀xεinf)<set,kind,grossk>=x; kind in kinds;
if atom set then map(set)=kind; else(∀yεset)map(y)=kind;; end if;
if grossk ne Ω then gross(kind)=grossk;; end ∀x;
  return map; end analyze;
/*next we analyze the user-supplied information defining the
left-and right precedence for each symbol kind, using a similar
format to that noted above. where one of these precedences
but not the other is stated, the same value will be used for
both when none is stated, the precedences of the gross kinds
will be used */ kinds=kindz;
```

```
lprec=analyze(lprinf); rprec=analyze(rprinf);
(∀x∈hd[lprec]|rprec(x) eq Ω) rprec(x)= prec(x);;
(∀x∈hd[rprec]|lprec(x) eq Ω) lprec(x)=rprec(x);;
(∀x∈kinds|lprec(x) eq Ω) lprec(x)=lprec(gross(x));
  rprec(x)=rprec(gross(x));;
/*now generate mask patterns for every kind of symbol;
  note use of user-supplied triple to control masking
  of three bits*/
/*pattern of 12 leftmost bits*/
symbtuple= <var,var,o,var,'(',var,')',{'(',er},var,{er,')'},
        {'(',er,o},{')',er}>
(∀x∈tℓ[symbkind] u tℓ[typkind] with 'var')
  mask(x) = gross(x) locsin symbtuple + x locsin triple;;
/*auxiliary routine to generate bit-vectors*/
definef setat locsin tuple; bitv=nul;
(while tuple ne Ω)<ent, tuple>=tuple;
bitv = bitv + if atom setat then
        if atom ent then setat eq ent else setat ε ent
        else if atom ent then ent∈setat else ∃x∈ent|x∈setat;
return bitv; end locsin;
/*here must follow user-supplied blocks, as follows:
block restart: gives any table-repair operations which must
  be executed on entrance to parser (might be null)
block usercode: all code necessary to handle user-specified
  exceptional situations (might be null);
block give(tokinf,typinf,lprinf,rprinf,triple,
  endfilesigns):
  tokinf- assigns a kind to each reserved token and
        a grosskind to the kind, if necessary;
  typinf- does same for lexical types having some special
        significance;
  lprinf,rprinf- assign possibly separate left and right
        precedences for each possible symbol kind;
  triple- may be 0o; describes sets of three symbols defining
        situations calling for exceptional actions;
endfilesigns - set of all lexical tokens defining illegal end-file
condition*/ /*here ends the entire preparser*/
end preparse;
```

We now wish to illustrate the use of the generalized precedence parser that has been described, and for this purpose will consider the problem of parsing a simple language somewhat resembling Dartmouth BASIC but rather more general. We take this language to have the lexical structure specified for the hypothetical language lexically somewhat like FORTRAN discussed as an example in section 4, cf. pp. 122 ff. The main features of the lexical structure of this language will be recalled by the following table, which lists all the lexical types of the language, and for each gives an example of the sort of pair which *nextoken* might return on encountering a token of this lexical type.

lexical type	sample pair returned by *nextoken*
integer	<irh,3> or <ir,3000>
real	<dip,3,> or <r,3.0>
name	<nm,boojum22>
special character	<nxt,+>
period delimited operator	<pdo,.exp.>
hollerith constant	<hc,etaionshrdlu>
end record	<er,nulc>
end file	<ef,er+er>

We shall take it as a property of our generalized BASIC that all keywords are reserved, and will therefore find it convenient to treat all keywords as operators. In addition to keywords, the following tokens will be used in special ways:

( ) : , > < + - * / .exp.

The last of these designates exponentiation; the 'greater than' and 'less than' signs are used to form comparison operators; and the colon is used to punctuate labels: any statement may be labeled; any valid name may be used as a label; a label must be followed by a colon.

A detailed formal definition of the language we have in mind will follow later. Here we shall only suggest the main outlines of the language by giving the following nonsensical program, which uses every form of statement which the language provides at least once.

```
LOOP:    READ A,B
         LET C(A)=A+A+F(A)
         IF A > B THEN LOOP
         FOR J = 1 TO 100 STEP 10
         FOR K = 1 TO 8
         GOSUB PROCESS
         NEXT K
         PRINT B,A
         NEXT J
         GO TO LOOP
PROCESS: READ C,D
         RETURN
         DEF F(X)=X.EXP.X
         DIM C(100), D(10,10)
         DATA 100,200,2.0,-2.0
         END
```

The keywords of the language fall into three groups:

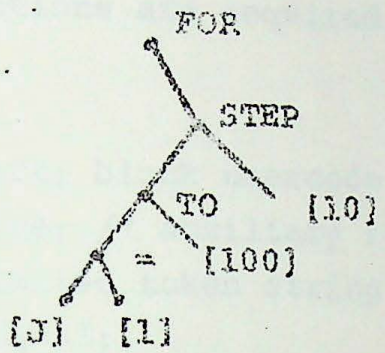
i. those which occur in 'first position' only:
LET, READ, PRINT, DATA, GOTO, GO, IF, FOR, NEXT, DIM, GOSUB, RETURN, DEF, END

ii. those which occur in 'second position'
TO, THEN

iii. STEP, which occurs in third position.

As we have already noted, our parsing strategy will involve the assignment of a precedence to every 'operator', which, of course, means to reserved key-words also. We wish to make these assignments in such a way that the preparser will develop a tree structure useful to the postparse process which is to follow. This means, for example, that keywords (which will

standard positions near the top of preparse-produced trees. E.g., the tree we develop for the "FOR-statement with STEP option" appearing in the sample program appearing above might with advantage be as follows:



Here, we have placed the "*nodtype*" of each non-terminal node next to the node, and, using square brackets, have indicated terminal nodes at which the *nodtype* function will not be defined. In order to assure that the trees produced by *preparse* have such a structure, we have only to assign suitable low precedences to keywords, with higher precedences going to expression separators like the comma, and highest precedences going to operators which can appear in arithmetic expressions. The following set of precedences (which, somewhat superfluously, allows for concatenations) will do nicely.

sign	left precedence	right precedence
(	1	10
'var', : let,read,print,data, goto,go,if,for,next, dim,gosub,return, def,end	2	2 3
step	4	5
to, then	5	6
=, comma, >, <	6	7
+, -	7	8
*, /	8	9
,exp,	9	1
)	10	

The conventions defined by the relevant details of the *preparse* routine given above allow us to create a preparser for the language we are considering by stating, in a suitably stilted form, the information contained in the preceding table. The user blocks needed are as follows; note that, since no user-defined preparse actions are required, the first two of these blocks are empty.

```
block restart; end restart; block usercode,end usercode;
/*both empty*/  block give; /* auxiliary routine forming
sequence from blank-delimited token string */
definef tilb(x); y=x; seq=nl;
(while 1 ≤ ∃[k] ≤ len y|(k elt y) eq ' ' or k eq len y)
<seq(#seq+1),y>=<if k lt len y then k-1 else k first y,
  (len y)-k last y>;
end while; return seq; end tilb;
tokinf={<tl[tilb(x)], tilb(x)(1),'o'>, xs{'step','to then',
 'let read print data goto go if for next dim gosub
 return def end', '= , > <', '+ -','* /', '.exp.', ':'}
u{< '(',')'>,<')',')'>,<nulc,er>,<er+er,er>};
typinf=nl; triple=0; /*no special lexical types or token sequences*/
lprinf={ <tl x, hd x>, xstilb{'( : let step to = + * .exp. )')};
lprinf('var')=2; rprinf={<'(',10>,<')',er},1>};
endfilesigns={<ef,er+er>}; end give;
```

This declaration gives a token-class and precedence structure to the generalized BASIC language which we illustratively consider.

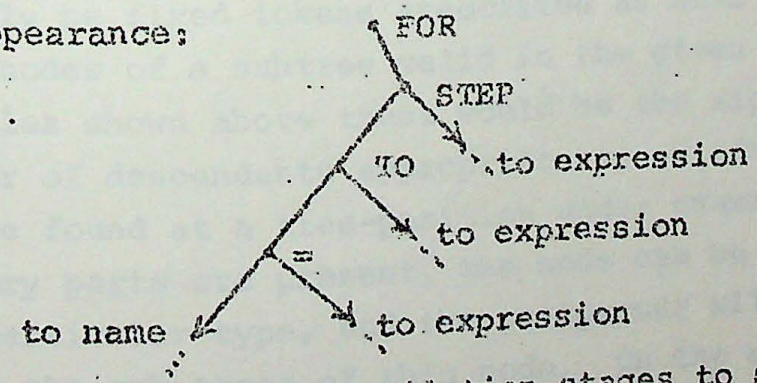
This is ample discussion of the general preparse process and of its application in a particular situation. We now go on to discuss the rather more interesting postparse process. In postparsing, the trees output by the preparser will be subject to systematic top-down verification, and, where necessary, reorganization using a standard diagnostic scheme, diagnostics aiding in the location of errors will also be produced.

If we think of the postparse process in top-down terms, its general outlines are simple enough. We start at the root of a parse tree produced by the preparser. By searching in the immediate neighborhood of the root, looking for key-words and key symbols in characteristic positions, we determine the type of statement with which we are dealing. Once this has been done, we know what kinds of subtrees are to be expected at each possible position relative to the root. We then go in turn to the root of each of these sub-trees, and, proceeding recursively downward, verify that each subtree is correctly formed. The general rule to be applied is always as follows:

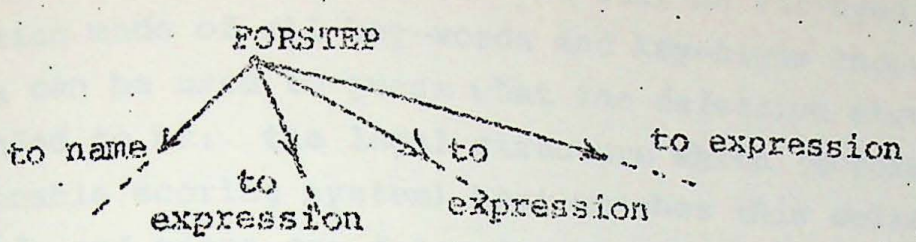
i. The configuration in the immediate neighborhood of a sub-tree root is legal;

ii. Each sub-subtree has (recursively) a legal configuration.

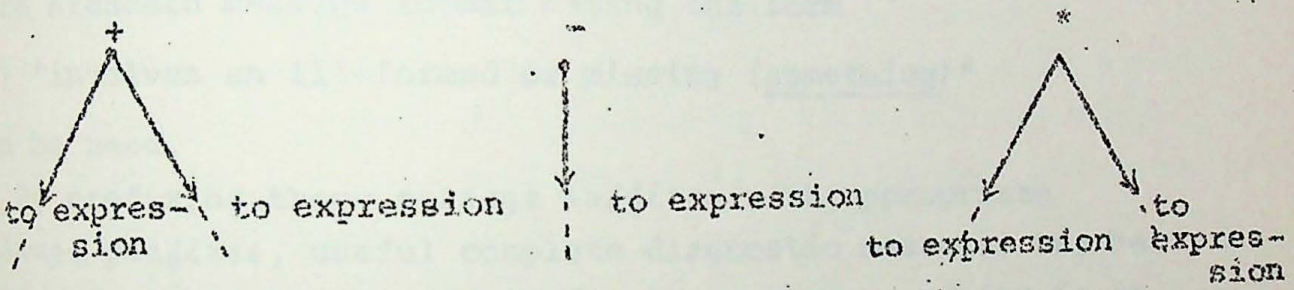
Various subsidiary processes associate themselves in a natural way with this recursive top-down structural verification scheme. In the first place, as we travel about the various portions of a tree verifying its structure, we may wish to reorganize the tree slightly. For example, if the precedences previously descended are employed, the parse tree for the BASIC FOR-statement with STEP-option will initially always have the following appearance:



It may be helpful to later compilation stages to simplify this structure, when it is recognized, into one which appears as follows:



Since we expect the postparser to detect errors in certain cases, we may readily enough require it to issue diagnostic messages in these cases. It is in a good position to do so, as it will always know both what it is looking for, and why it failed to find what it is looking for. For example, if an *expression* is required in a certain context, nodes like



will be valid, others not. Certain *characteristic tokens* will be associated with each node type valid in a given context; these will generally be fixed tokens associated as node-types with the top few nodes of a subtree valid in the given context. In the examples shown above these would be the signs +, -, *, etc. If the number of descendants appropriate to an admissible node-type are found at a tree-position under examination, and all obligatory parts are present, the node can be accepted as being of a particular type, and the postparser will move down to deal with the sub-trees of this node. On the other hand, when none of the node types allowed in a given context match the pattern of subparts actually found at a node being processed, an error has been detected.

In this case, the postparser will prepare a helpful diagnostic message, as follows. First, the nodes immediately below that

at which the error has been detected will be surveyed, and a collection made of all key-words and key-signs encountered. This data can be used to guess what the defective structure was intended to be: the legal structure which (according to any reasonable scoring system) best matches this collection of keywords and signs can determine the guess made. Once this is done, part of a standard diagnostic message will be built up; a reasonable form for this is as follows:

(1) "is probably an ill-formed (something) but (some symbol) is found where (other symbol) is required, (some symbol) is found where (other symbol) is required, ..."

If no legal structure giving a plausible match is found, a more standard message format having the form

(2) "involves an ill-formed or missing (something)"

can be used.

By prefixing these *message suffixes* with appropriate *message prefixes*, useful complete diagnostic messages can be produced. The scheme used to compose a message prefix is as follows: As the postparser proceeds to lower levels of a tree, it will build up a stack describing the chain of nodes through which it has passed, listing these nodes by type, and, when necessary, by serial number among all nodes of a given type sharing a common ancestor-node. This information will then generate a message prefix having the following general form:

(3) the (n-th) (something) of the (n-th) (something) ... of this something .

A typical diagnostic-message might then be

'the third factor of the second expression of this for-statement involves an illegal or missing operator, or perhaps some illegal token used to represent a quantity'.

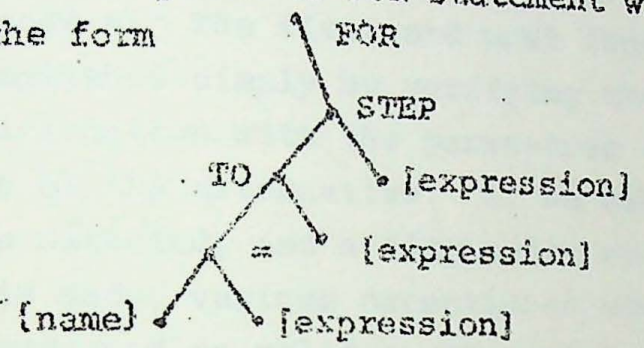
The utility of these rather detailed diagnostics can be improved by adding the following simple trick to our approach. The lexical

scanner can associate a starting and an ending position (in its *source statement*, i.e. the character string presented to it) with each token it generates. This data will carry through the preparser, and can be used to associate a starting and ending position (again in the source statement) with each subtree of the parse-tree developed by the preparser. Then when an error is detected it will always be localized to a certain subsection of the source statement. If we number diagnostic messages as they are generated, we may then print a copy of the source statement in our output listing, underprinting each defective subsection with the serial number of a relevant message. (These serial numbers may be interspersed with asterisks, for vividness). This will in many cases pinpoint the cause of a particular error.

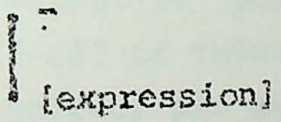
We will shortly give a SETL program for the postparser whose general character has been outlined. All the information concerning the tree structures admissible in a particular language will be made available to the postparser by providing a descriptive data-structure, which may be called an *expanded tree-grammar*, to it. This expanded grammar will also specify the information which is necessary to determine the actions of the postparser in dealing with parse trees which contain errors. As these tree-grammars will provide facilities for the description of a large variety of contingencies, they may, at their most complex, be highly structured; the postparser algorithm which follows therefore requires a certain measure of preliminary explanation, which we now proceed to give.

Note first of all that our tree-grammars like the more familiar class of context-free grammars, are constructed in a manner reflecting two basic notions: the notion of *syntactic type* and the notion of *alternative realizations* of a syntactic type. A syntactic type is essentially defined by the set of alternative realizations associated with it; we call these its *alternatives*.

Each alternative is in effect a small tree, to the nodes of which *types* are associated by a *nodetype* function; this tree describes the structure which the vicinity of a node in a preparse-produced tree must have if it is to be accepted as realizing a given alternative of a given syntactic type. For example, in the complete tree-grammar for generalized BASIC to be described later in the present section, one alternative of the syntactic type *statement* (namely, that alternative which corresponds to 'FOR-statement with STEP option') has essentially the form



One alternative of the syntactic type *expression*, namely that corresponding to the manadic use of the 'minus' sign, has essentially the form



These diagrams show some of the characteristic features of alternatives in our tree-grammars.

i. Nodes of an alternative will either be *terminal* (twigs) or *non-terminal* (intermediate nodes).

ii. Each node has a *node-type*. The type of a non-terminal node will be some set of literals (keyword or keysigns, treated in the preparser as operators). The type of a terminal node may be either

a) a syntactic type (indicated in the diagrams just above by square brackets)

b) a set of admissible lexical types (indicated in the above diagram by curly brackets)

c) a set of literals which may be required, possibly with a specified lexical type.

Alternative c) would cover the case of non-reserved keywords treated by the preparser as quantities rather than as operators.

The postparser will travel about a preparser-built tree, verifying and modifying its structure. At each moment, it will be aware of a given context and will be attempting to match a given node n of the parse tree to a syntactic type admissible in this context. The most basic step in this process will be the discovery of some alternative of the syntactic type whose tree structure matches the parse-tree structure in the immediate vicinity of the node n. The first and most fundamental step of matching is accomplished simply by verifying the structural identity of an alternative with the parse-tree vicinity of n, down to the twigs of the alternative. If no match is possible, an error has been detected, and a diagnostic routine is entered. Even if a match is made, various exceptional user-supplied tests may have to be performed to validate the match. If all of these tests (and in most cases none will be required) are successfully passed, then the postparser will descend to lower tree levels and there repeat its actions of verification and modification. When it returns successfully from these lower levels it may apply various additional user-supplied "post-tests" for final verification of the match. If all of these tests (and again in most cases none will be required) are successfully passed, a given part of a parse-tree structure will have been validated. This portion can then be restructured in any manner that may be necessary, following which the postparser may return successfully to the parent node of n.

As the preceding paragraph begins to make plain, our tree grammar will associate not only the fundamental tree-like structures described above, but also various additional information items, with an alternative. Some of these items may be programmed auxiliary actions and tests. Such auxiliary actions as may be specified by the user will be formed, in the manner described in section 4, into a package of *auxiliary routines* called *apak*; see the lexical scanner code given there

and its *setup* routine for details. The tree-grammar will associate with each alternative, identifiers of any auxiliary actions which the alternative may require. In more detail, we allow auxiliary actions of the following types to be associated with an alternative:

iii. Pretests: These are programmed functions, returning a value t or f; if any returns the value f, the alternative will in the manner described above be rejected as a match for whatever parse-tree portion is being examined.

iv. Preactions: These are programmed actions to be performed after all pretests have been passed but before the postparser descends from a given tree-portion to its various subtrees.

v. Post-tests: These are programmed functions, returning a value t or f; if any returns the value f, the alternative will, in the manner described above, be rejected *a posteriori* as a match for whatever parse-tree portion is in question.

vi. Post-actions: These are programmed actions to be performed after all post-tests have been passed, but before the postparser returns to the parent node of a node with which it has been concerned.

As we have noted, optional information of the categories iii-vi will generally be absent.

We have already remarked that, after verifying the admissibility of a sub-tree structure, we may wish to modify it in one or another manner facilitating the later parts of a total compilation process. In order that this should be possible within the general logical framework defined by our postparser, we allow the association with an alternative of

vii. A re-evaluation specification: If none is given, the restructured subtree to be associated with a parse-tree node is obtained merely by restructuring the subparts of the tree dependent from this node. On the other hand, if a re-evaluation specification is given, the restructured subtree to be associated with a node will be obtained by combining certain of these subparts with other tree-structures to be calculated by user-supplied programmed functions.

We noted in our description of the preparse algorithm that when a series of syntactic entities occurs repeatedly separated by a fixed operator sign, the preparser will attach a series of subtrees representing these entities to a single node representing the operator sign. Where this construction is used within a language, our tree-grammar for the language will reflect it. For example, the READ statement in BASIC may involve a list of (possibly indexed) variables separated by commas; the corresponding alternative of the syntactic type *statement* in our tree grammar has the form

```
read
, (2)
[indexed variable]
```

The number n appearing in parentheses is a *multiplicity*, and its appearance designates the branch below it as a *multiple descendant*; the value of n defines the minimum number of copies of this branch acceptable if the alternative is to be matched to the neighborhood of a parse-tree node under consideration.

To match an alternative to a portion of a parse tree, we will always verify that the types of all intermediate and all 'literal' twigs of the alternative are matched in the parse tree; and if a 'lexical' twig occurs will always verify that a token of the appropriate sort occurs at the correct position of the parse tree. However, where syntactic types designating possibly compound substructures occur at the twigs of a parse tree, we will not normally examine these structures before accepting the match. (Of course, these structures will in any case be examined after the match is accepted. A failure after this acceptance will however lead to the production of a diagnostic rather than to an attempt to find some other matching alternative.) In certain exceptional circumstances, however, we may wish to vary this procedure, and to examine substructures before accepting a match. Substructures for

which this is to be done are called *obligatory*, and the corresponding twigs within an alternative are specially marked.

This completes our account of that information which may be associated with an alternative and which will be used even if no error is detected by the postparser and no diagnostic need be produced. However, for use in the production of diagnostics, additional alternative-associated information will be provided. This will include

viii. An external name to be used in diagnostics when it is necessary to refer to the alternative;

ix. A comprehensive collection of all the fixed tokens which appear at the intermediate nodes and twigs of an alternative, each being given with an indication of its tree-position in the alternative at which it appears. With each of these tokens we also associate a *scoring factor*, which might also be called the *evidential weight* of the token. When we are unable to match any acceptable alternative to a section of a parse tree, we will use these weights to guess what structure the unmatchable structure actually found was intended to represent. This guess will be made according to a highest-total-score scheme, tokens found in correct tree-positions being counted with double score. If the highest score found by this matching process exceeds a standard minimum *threshold*, a corresponding guess will be made, and the diagnostic message which is issued will be structured accordingly. If the highest score attained is too low, a standard diagnostic fragment, determined by the syntactic type sought within a given context, will be used instead.

This finally gives a sufficiently complete account of the information which a tree-grammar may associate with an alternative.

A tree-grammar will also associate various information items with a syntactic type. These are as follows:

i. A set of all the lexical alternatives which might, as single tokens, be acceptable alternatives for the syntactic type;

ii. An external name to be used in diagnostics when it is necessary to refer to the syntactic type;

iii. A keysymbol name to be used in diagnostics when it is necessary to complain about items missing from a given syntactic type;

iv. A successor type, if when no alternative of a given syntactic type is matched an item of some other syntactic type would be equally acceptable. (For example, in many languages 'statements' are acceptable where 'labeled statements' cannot be found.)

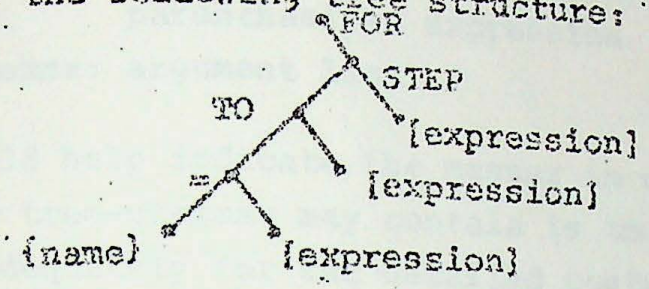
This information is fundamental. However, additional information is provided in order to improve the efficiency of our top-down process. This additional information is as follows.

v. If a syntactic type and the type of a parse-tree node to be matched to it together determine a small set of possibly matching alternatives, we record this set. Note, for example, that if we seek the syntactic type 'expression' and see a node of type '*' then only the 'product' alternative of 'expression' can be in question.

vi. If a syntactic type and the type of a parse-tree node n to be matched to it together determine only a rather large set of possibly matching alternatives, but the presence of a particular key-token as the j-th descendant of n will reduce this set considerably, we record j. In this case, we also record the sets of alternatives which are associated with the occurrence of particular key tokens in this characteristic position. Note, for example, that in dealing with the syntactic type 'statement' in languages with non-reserved key-words we may wish to use this feature; often the presence of certain keywords in the first position of a statement will tell us what alternative ought to be considered.

This concludes our survey of the information which a tree-grammar may associate with syntactic types and their alternatives. In order to make this rather long list of specifications more

comprehensible, we give some examples. As has been noted, the alternative 'FOR-statement with STEP option' of the syntactic type 'statement' in our extended BASIC is described by the following tree structure:



The additional information associated with this alternative is as follows.

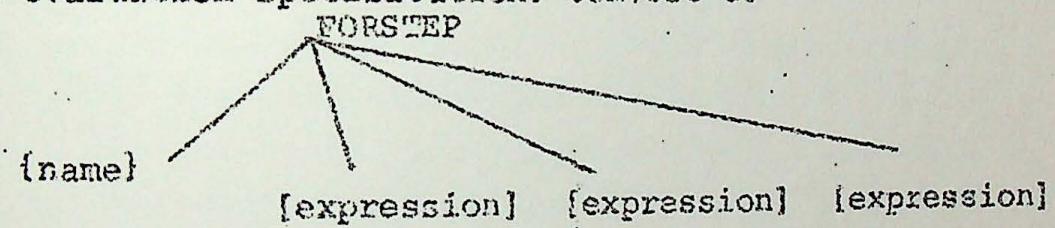
iii. Pretests: none

iv. Preactions: none

v. Postests: none

vi. Postactions: none

vii. Re-evaluation specification: convert to



Multiplicities: none. Obligatory compound subparts: none.

viii. External name: 'for statement'

ix. fixed tokens, with their scores:

FOR-scores 6; STEP-scores 6; TO-scores 4.

For the syntactic type 'expression' we would record the following information.

i. Admissible lexical alternatives: name, integer, real number

ii. External name: 'expression'

iii. Keysymbol name for complaint in diagnostic:
'operator or perhaps some illegal token used to represent a quantity'

iv. Successor type: none.

v. Alternatives as determined by type of node to be matched:
+: dyadic sum, monadic positive quantity
-: dyadic difference, monadic negative quantity

*: product

/: quotient

.exp.: exponential

( ) : indexed variable or function call,
parenthesized expression

comma: argument list

This should help indicate the manner in which all that information which a tree-grammar may contain is to be used. To prepare still more adequately for the detailed postparse algorithm which is to follow, we now give a table which summarizes what has been said in the last few pages, and which also notes the function names under which various information items will appear in our SETL algorithm, and the arguments which must be supplied to these functions.

Data and functions supplied by the postparse setup block.
(Dramatis Personae of the *postparse* algorithm)

A. Functions Associated with the alternatives of a syntactic type

1. pretests (alt) - a tuple of identifiers for t- or f- returning programmed functions.
2. preactions (alt) - a tuple of identifiers for preactions to be performed.
3. postests (alt) - a tuple of identifiers for t- or f- returning programmed functions. Each component of this tuple is actually a pair, whose first part identifies a function, and whose second is a message-fragment to be printed if the test fails.
4. postacts (alt) - a tuple of identifiers for post-actions to be performed.
5. reval(alt) - a tuple which, if present, specifies the manner in which a tree-section is to be restructured. The first component of this tuple determines the node-type of the new structure; it may either be a fixed literal or the identifier of a user-supplied function which will produce a literal. All remaining components are pairs, having the following possible forms and significances:
   <0,identifier> : identifies a programmed function producing a subtree portion;
   <locator,0> : identifies a part of an original parse tree which is to be used within a reorganized tree (this subpart may also have been reorganized);
   <locator,start> : identifies a possibly multiple part of an original parse tree which is to be used within a reorganized tree.
6. minlast(alt) : defines a minimum multiplicity required for the last descendant of a given alternative; if undefined, alternative is not multiple.
7. oblig: a set defining all those twigs of an alternative which represent *obligatory* composite types.

8. literals: a set defining all those twigs of an alternative which represent specific literals or sets of literals.

9. lexics: a set defining all those twigs of an alternative which represent specific lexical types or sets of lexical types.

The node-type of an intermediate node in an alternative will be a set of tokens: those acceptable for a match. The node-type of an alternative-twig will generally be a syntactic type. However, if the twig belongs to *lexics*, the value of its node-type will be a set of lexical types (namely, those acceptab... as a match to the twig); if the twig belongs to *literals*, the value of its node-type will be a set of tokens (those acceptable as a match to the twig). Moreover, if the function

10. lextype(alt) is defined for a literal twig, it specifie... the lexical type which these tokens must have to be acceptable as a match to the twig.

11. altname(alt): the external name of an alternative.

12. gathalt(alt): a set defining the fixed tokens characteristic of an alternative, their tree-positions, and their evidential weights.

13. threshold: the minimum score needed to justify a specific guess.

14. fixed(twig): an integer defining the serial number of a given twig among all twigs of an alternative which correspond to composite substructures of a given syntactic type. This information is used when producing diagnostic messages.

B. Functions associated with a syntactic type

15. lexalts(synt): the set of all lexical alternatives which might be acceptable instances of the syntactic type.

16. namesynt(synt): the external name of the syntactic type.

17. keysymbol(synt): a string to be used in diagnostics when no specific guess concerning an intended alternative is justified.

18. sesor(synt): a successor type, acceptable in place of a given syntactic type.

19. altset(synt,nodtype): the set of all alternatives of a given syntactic type which might possibly be matched to a syntax-tree node of given type.

20. lockey(synt,nodetype): the location in which a literal useful in the selection of a set of alternatives can be found.

21. secaltset(synt,literal): the set of all alternatives of a given syntactic type which might possibly be matched to a syntax node having a specified literal in a characteristic position.

C. Functions of general use

22. nodtype(alt): the node type of an alternative. See the remarks following item 9 above.

23. desc(alt,j): the tree-structure-defining descendant function.

24. xpak(act): invocation of a user-supplied routine by its identifier.

D. Additional data objects, not supplied by *setup* routine

25. mstak: a stack of pairs, the first component of each pair being a number, the second a syntactic type name; used to produce prefixes of diagnostic messages.

26. nodnum: set to -1 to signal that no diagnostic messages or tree-revising actions are required; otherwise gives the serial number of the syntactic type currently being sought, among all twigs which are part of a single parent alternative.

27. value: attaches a revised tree structure to an old node during tree reorganization.

28. partseq: a tuple enumerating the composite syntactic types attached to the twigs of an alternative, and the parse tree nodes to which these refer. Each component of this tuple is either a pair, or, in the case of twigs with multiplicity, a triple whose last component is an integer.

This completes our account of the information items used in the postparse algorithm which follows. We shall make only a few additional remarks before giving the SETL code for this algorithm. First, a detail: the tuples used in the algorithm will be of one of two types, as appropriate. Some will simply be tuples of atoms. Others will have ordered pairs or tuples as components. In such cases, to avoid confusion, we include the null-set as a terminating final component; a tuple terminated in this way might be called *null-terminated*. (This is also the device used to structure lists in LISP.)

Next, a more significant point. Generally, the alternatives of a syntactic type of a tree-grammar will be unordered and matches can be attempted in any order. In some special cases, however, we may wish to attempt to match certain alternatives in a fixed order, so that before we try one alternative we can be sure that some earlier alternatives have failed. Alternatives related in this way are said to form a *sequence of alternatives*. Tree-grammars will allow such sequences, and the algorithm which is to follow will reflect this fact.

Finally, an observation. A tree grammar, with all its parts and options, will be a rather complex data object and it might be tedious to set one up if this had to be done directly. Such is of course not our intent. We shall in fact provide a specialized input format for the description of tree-grammars; this format somewhat resembling the *Backus normal form* regular used for the description of languages, but with extensions to accommodate all the additional information items, diagnostic and otherwise, which are to be supplied. Data sets having this format may be regarded as programs written in a specialized *postparse metalanguage*. The digestion of material having this input format, and the transformation of this material into a set of mappings having the forms and significances described above, is the work of the *setup* block associated with the postparser.

As the structure of such a setup block should be sufficiently clear from the example given in section 4, we shall leave it to the reader to supply the code constituting this block, and (after having given the postparser algorithm) shall content ourselves with describing the data formats to be used. Note, however, that the necessary setup procedures can be created most easily as follows: using data having the standard SETL 'read' format build up a tree grammar for the postparse metalanguage. The necessary setup routine will then consist of a copy of the postparser to which this grammar is attached.

As will be seen below, the postparser, addressed in this way by use of a postparse metalanguage, is a very powerful tool for the succinct and convenient definition of new programming languages. Using it we can write compiler 'front-ends' with a minimum of difficulty; since it allows arbitrary reorganizations to be applied to a parse tree, we can with some effort even use it to generate machine code.

But, after all these preliminaries, it is now time to give the detailed SETL code for the postparser.

```
definef postparse(syntype,node,nodenum);
/* this routine is written as a function which returns t if
   a valid parse is found, f otherwise. we have also nodnum=-1
   if a preliminary test only is being made, and no diagnostic
   messages are to be emitted or subtree values are to be
   calculated, otherwise nodenum=0 if the given node is the
   only element of its type in the context being considered;
   and nodnum=j > 0 if the given node is the j-th element
   of its type in the context being considered */
/* declaration of the various tables used, and of other external
   data objects */
parser external lockey,altset,secaltset, sesor,threshold,
   namekeyelt, namesynt, pretests, preacts, postests,
   postacts, rpak, reval, gathalt, minlast, fixed, oblig,
   literals, lexics;
parser external mstak, value, desc, nodtype;
initially do setup; mstak=nil; /*initialization block to supply
   all sets (i.e. tables) and auxiliary routine package rpak
   needed below. details of this block will not be given */
synt=syntype;
[syntry:] /*fast key-search process to determine subset of
   alternatives to be examined in detail*/
if nodtype(node) eq 0 /*so that node is terminal*/
   then if(setalts is lexalts(synt)) eq Ω
   then go to maynext; else go to findalt; end if nodtype;
if(keyloc is lockey (synt, nodtype(node))) eq Ω
/* in this case, the node-type itself will serve to determine
   the set of alternatives to try */
then if (setalts is altset(synt, nodtype(node))) ne Ω then
   go to findalt; else go to maynext;;
else /*key is either node-type of specified descendant,
   or is literal token */
```

```
if(keydesc is desc(node,keyloc)) eq Ω /*required key not present*/
  then go to maynext;; key=if(x is nodtype(keydesc)) ne Ω
  then x else tℓ keydesc;
if(setalts is secaltset(synt,key)) ne Ω then go to findalt;;end else;
[maynext:] if(x is sesor(synt)) ne Ω then synt=x; go to syntry;;
/* otherwise try guessing process if appropriate, and return
  one or another type of error message */
if nodnum lt 0 then return f;;
guess(node,syntype,score); if score lt threshold then
  print prefix(synt, nodnum) + 'involves an ill-formed or missing'
                + namekeyelt(synt);; return f;
[findalt:]  parseq = nℓ;
  if ∃[alt] ∈ setalts|atom alt and matches (alt, node,partseq)
                then go to matched;;
  if n ∃[altseq] ∈ setalts, 1 ≤ [j] ≤ if atom altseq
                then 0 else #altseq|
          matches(altseq(j),node,partseq) then go to maynext;;
alt   = altseq(j);
[matched:] /*here we have found a matching alternative, and are
  prepared to descend in the tree. the matching process will
  also produce a list 'partseq' of nodes to be explored
  recursively, a node itself being given if no repetition
  is involved, but its parent node and the number of the
  starting descendant being given if repetition is involved*/
/* do all pre-actions associated with the alternative */
tup=preacts(alt); (while tup ne Ω) <act,tup>=tup; rpak(act);;
/* now verify subparts */
if nodnum ge 0 then mstak(#mstak+1)=<nodnum,namesynt(synt)>;;
ok=t; typecount=nℓ;
(while partseq ne nℓ)<part,partseq>= partseq;
<subsynt,subnode,start>=part; if start eq Ω then
/*case of non-multiple node*/
subnum = if nodnum lt 0 then nodnum else
  if (x is fixed(subnode)) ne Ω then x else
  if (tc is typecount(subsynt)) ne Ω then tc+1 else 1;
```

```
if nodnum gt 0 and x eq Ω then typecount(subsynt)=subnum;;
ok=ok and postparse(subsynt,subnode,subnum);
else /*case of multiple node */
(start ≤ ∀j ≤ #desc(node))
subnum=if nodnum lt 0 then nodnum else
  if (tc is typecount(subsynt)) ne Ω then tc+1 else 1;
if nodnum gt 0 then typecount(subsynt)=subnum;;
ok=ok and postparse(subsynt,desc(subnode,j),subnum);
end ∀j; end else;
/* if failure, need not try sny post-tests or actions,
   simply return */
if n ok then return f;;
/*otherwise do all post-tests, and, if appropriate, post-action
tup=postests(alt); (while tup ne nℓ) <<act,msg>,tup>=tup;
rpak(act); if ok eq f then if nodnum ge 0 then
  print prefix(synt,nodnum)+msg;; return f; end if ok; end whil
if nodnum lt 0 then return t;; /*otherwise do post-actions
   and value calculations */
tup=postacts(alt); (while tup ne Ω)<act,tup>=tup; rpak(act);;
if(revtup is reval(alt)) eq Ω /*so that no re-evaluation is
  necessary*/ then value(node)=node; if nodnum ge 0
   then mstak(#mstak)=Ω; return t; end if;
/*otherwise re-evaluate as indicated by the components of the
  re-evaluation tuple. this is nℓ-terminated, and has the form
<new type,<locator, action or start>,<locator, action or start>
the detailed conventions used for each type of component will
be seen in the code which follows */
new=newat; <type,revtup>=revtup;
if n integer type/* so that a literal is given*/ nodtype(new)=type;
   else /*type is computed by a user-supplied  function*/
   rpak(type); nodtype(new)=val;;
/*now attach specified descendants to new node*/
(while revtup ne nℓ) <<locator,actstart>,revtup>=revtup;
   if locator eq 0 /*so that a calculation is to be performed*/
   then rpak(actstart); val attach new; else if actstart eq 0
/*so that a non-multiple part is involved */  then
```

```
value(partof(node,locator)) attach new;
else /* a multiple part is at hand */
(actstart ≤ ∀k ≤ #desc(parent is partof(node,locator)})
value(desc(parent,k)) attach new;; end else;
value(node)=new;
if nodnum ge 0 then mstak(#mstak)=Ω;;
return t;
/*this is end of main body of postparse routine.
  now follow various subroutines called */
define piece attach node; postparse external desc;
/*auxiliary tree-build routine */
desc(node,#desc(node)+1)=piece; return; end attach;
definef matches(alt,node,partseq);
/*key routine to test vicinity of tree against pattern and
  pretests specified by a tree-grammar alternative */
postparse external pretests, rpak;
if n matcher(alt,node) then partseq=nl; return f;;
/* else apply pretests */
tup=pretests(alt); (while tup ne nl) <act,tup>=tup;rpak(alt);
if n ck then return f;; end while; return t;
definef matcher(alt,node); /*inner recursive routine */
matches external partseq; postparse external desc,nodtype,
  minlast,literals,lexics,oblig,lextype
if desc(alt,1) eq Ω then go to twig;;
/*else not twig; type and subparts to be examined */
if (nt is nodtype(node)) eq Ω then return f;;
  if n nt ∈ nodtype(alt) then return f;;
desreq=(ndesca is #desc(alt))+if(min is minlast(alt))
  ne Ω then min-1 else 0,
/*check on appropriate number of descendants */
if(ndesc is #desc(node)) lt desreq or (min eq Ω and ndesc
  ne desreq) then return f;;
/* otherwise check parts individually */
if 1 ≤ ∃j ≤ ndesca -if min ne Ω then 1 else 0 |
  n matcher(desc(alt,j),desc(node,j))
  then return f;; if min eq Ω then return t;;
```

```
/* otherwise last is multiple;check on whether obligatory or not*/
if (descalt is desc(alt,ndesca)) ε oblig
   then /*obligatory-check subparts first */
if ndesca ≤ ∃j ≤ ndesc|n postparse(nodtype(descalt),
   desc(node,j),-1) then return f;;
/*obligatory subparts all checked */  end if;
/* make addition to partseq */
partseq=<<nodetype(descalt),node,ndesca>,partseq>; return t;
[twig:] if alt ε literals then return <*z2> node ε  nodetype(alt)
          and  if(lext is lextype(alt)) eq Ω then t
          else hd node eq lext;;
       if alt ε lexics then return hd node ε  nodtype(alt);;
       if alt ε oblig then /*obligatory-must check */
          if n postparse(alt,node,-1) then return f;; end if alt;
       /* make addition to part sequence */
    partseq=<<nodtype (alt),node>,partseq>; return t; end matcher;
end matches;
definef partof(node,locator); postparse external desc;
/* auxiliary routine to find node at a given tree address */
nod=node; (while locator ne nl) <adr,locator>=locator;
nod=desc(nod,adr);; return nod; end partof;
definef prefix(syntype,nodnum);
/*makes up message prefix from stacked name information*/
postparse external mstak,namesynt;
return 'the' + if nodnum eq 0 then nulc
  else(dec nodnum+ affix(nodnum)) + namesynt(syntype) + 'of'
   + [+: #mstak ≥ j > 1]('the' +
if(h is hd mstak(j)) eq 0 then nulc else (dec h + affix(h))
  + tl mstak(j) + 'of') + 'this' + tl mstak(1);
end prefix;
definef affix(n); /* an output nicety */ return if n %e 3 then
  {<1,' "st">,<2,' "nd">, <3,'"rd'>}(n)
else '"th'; end affix;
```

```
define guess(node,syntype,score);
/* routine to guess likeliest construction and print message
   when no precise match is found */
postparse external sesor, threshold, nodnum, altname, desc,
   altset,secaltset;
/* first gather information about top part of tree */
gather = {pickup(node)};
(1 ≤ ∀j ≤ #desc(node)) nodel =desc(node,j);
gather=gather with <pickup(nodel),j>;
(1 ≤ ∀k ≤ #desc(nodel)) gather=gather
   with <pickup(desc(nodel,k)),j,k>;
end ∀k; end ∀j;
score=0; synt=syntype; /*explore syntype and all its successors*/
(while synt ne Ω doing synt=sesor(synt);)
/*form set of possible alternatives for this syntactic type*/
allalts=[u: x∈t&[altset(synt)]] u [u: x∈t&[secaltset(synt)]];
(∀altcallalts) if(newscore is fit(alt,gather)) gt score
   then synkeep=synt; keep=alt; score=newscore; end if;
   end ∀alt; end while;
if score &t threshold then return;;
/*else have guessed alternative. prepare message, first part */
message=prefix(synkeep,nodnum) + 'is probably a'
   + altname(alt) + 'but';
/*call routine to prepare message, second part */
failmatch(alt,node); print message; return;
definef pickup(node); postparse external desc, nodtype;
/* auxiliary routine to obtain appropriate fragment of tree node*/
return if(x is nodtype(node)) ne Ω then x else <*z 2> node;
end pickup;
end guess;
```

```
definef fit(alt,gather);
/* scoring routine which matches elements of an alternative
   to an evidential set */
gath=gather; totscore=0;
/* 'gathalt' is associated with 'alt' by a part of the setup
   block; this part is similar in action to the subroutine
   'gather' appearing above. its value is a set whose elements
   have the structure
     token,position relative to the top of 'alt' ,score  */
(∀x ε gathalt(alt)) if ∃[g]εgath|(hd x) eq g then
   totscore=totscore+(2*tl x);gath=gath less g; end if; end ∀x;
(∀xεgathalt(alt)) if ∃[g] ε gath|(hd hd x) eq hd g then
   totscore=totscore + tl x; gath=gath less g; end if; end ∀x;
return totscore; end fit;
define adm(string);  guess external message;
/*auxiliary routine to add a terminal section to a message */
charad=if(3 last message) eq 'but' then ' ' else ' , ' ;
message=message+charad+string; return; end adm;
define failmatch(alt,node);
/* diagnostic producer which follows the action of 'matcher',
   producing messages instead of error-signals */
postparse external desc, nodtype,oblig,minlast,literals,
   lextype, lexics;

if desc(alt,1) eq Ω then go to twig;;
if(ndesc  is #desc(node)) eq 0 then ndesca=#desc(alt);
   go to numwrong; end if;
if n(ntn is nodtype(node)) ε (nta is nodtype(alt)) then
   adm(ntn + 'found where' + ntn + 'is expected');
   return; end if;
```

```
ndesca=#desc(alt); mult=minlast(alt);
(1 ≤ ∀j ≤ (ndesca-if mult ne Ω then 1 else 0) min ndesc)
  failmatch(desc(alt,j),desc(node,j));;
/*otherwise last is multiple; check whether obligatory or not*/
if(descalt is desc(alt,ndesca))coblig then
/* check obligatory subparts */
(ndesca ≤ ∀j ≤ ndesc)n postparse(nodtype (descalt),desc(node,j),-1)
adm(dec(j-ndesca+1)+ affix(j-ndesca+1)+ 'part found not to be'
  + namesynt(nodtype (descalt)) + 'as required'); end ∀j;
/* check on number of parts */ {numwrong:}
if ndesc lt ndesca then adm(if mult ne Ω then 'at least'
  else nulc + dec(ndesca-ndesc)+ 'parts missing'); return; end if;
/*else enough parts. check whether too many */
if mult eq Ω and ndesc gt ndesca then
  adm(dec(ndesc-ndesca) + 'superfluous parts'); return; end if;
/* multiple node case; check on presence of required minimum*/
if ndesc lt (ndesca+mult-1) then
adm('only'+dec(ndesc-ndesca+1) + 'parts found where'
  + dec mult + 'are required');; return;
{twig:} if altsliterals or altslexics and desc(node,1) ne Ω
  then go to falsetwig;;
  if altsliterals and n(litfound is <*z2> node) ∈
  (litneed is  nodtype(alt)) then adm(litfound+'found where'
  + litneed + 'or equivalent is required'); return;;
  if altsliterals and (lext is lextype(alt)) ne Ω and
        n(typef is hd node) eq lext then
  adm('literal'+litfound+ 'of lexical type' + typef
    +'found where lexical type'+lext+'is required'); return;end if
  if altslexics and n(typef is hd node) ∈ (lext is nodtype(alt))
    then adm('lexical type'+typef+'found where'+lext
    +'or equivalent is required'); return; end if;
  if altscoblig /*obligatory - must check */
and n postparse(alt,node,-1) then
adm('part found not to be'+namesynt(nodtype(alt))+'as is required'.
end if; return;
```

```
falsetwig: /*composite structure found where
                terminal is required */
req = if alt & literals then 'literallike'+ nodtype(alt)
      else if altslexics then 'lexical type' + snodtype(alt)
        + 'or equivalent';
adm('composite structure containing' + snodtype(node)
        + 'found where' + req + 'is required'); return;
end failmatch;
/* */  end postparse;
/* end of entire postparse package */
/* */
```

As we have already noted, the task of setting up a tree-grammar without any mechanical aids might be onerous. We shall therefore specify tree-grammars in a special format chosen so as to lighten this burden. The conventions descriptive of this input format may be thought of as defining a postparse meta-language. We shall now describe this metalanguage; we leave it to the reader to design a *setup* routine capable of transforming a block of statements in the metalanguage into the various mappings and sets required by *postparse*.

The skeleton of our format is this:

```
syntactic type a = alternative 1
                 = alternative 2
                 = alternative 3
                 ...
syntactic type b = alternative 4
                [= alternative 5
                 =]alternative 6
                 ...
```

with syntactic type names appearing on the left and all the alternatives of a given syntactic type appearing on the right. Where several alternatives are to form a *sequence* within which earlier alternatives are always to be attempted before later alternatives are examined, the group of equal-signs prefixing these alternatives will be included in square brackets, in the manner indicated.

An alternative is always represented by a sequence of *parts*, of which the first three are generally obligatory, while the others are optional. If all optional parts were actually included, the general appearance of an alternative would be as follows:

(1) tree-describer/name-specification/evidential weights of literals/ob/obligatory nonliterals/t1/pretests, separated by commas /a1/preactions, separated by commas /t2/ postests, separated by commas /a2/postactions, separated by commas /rv/ re-evaluation specification /rs/ resolution specification

As indicated in the above, optional parts are prefixed by a characteristic token included between slashes; they may consequently appear in any order.

For the moment, we postpose outlining the detailed structure of a tree-describer. A name-specification is a string of symbols (not including any unquoted 'active' symbols, as e.g. slashes or periods), which defines the external name of an alternative. Thus it might for example appear as

/product/

However, the name specification of the *first* alternative of a given syntactic type has the somewhat expanded form

/syntactictypename.keysymbolname.alternativename/ ,

i.e., gives three token strings separated by periods, of which the first is the external name of the syntactic type itself, the second is the key-symbol name of the syntactic type, and the third is the external name of the alternative. If the external name of a syntactic type is the same as its internal name, it may be omitted. If the external name of the first alternative of a syntactic type is the same as the external name of the syntactic type, it may be omitted. However, even when names are omitted, all punctuation (periods and slashes) must remain.

Evidential weights are integers, one of which must be given for every literal (or literal-set, see below) which appears in the tree-describer part of an algorithm.

Next we consider the form of a tree describer. A tree description consists of a sequence of *items*, separated by blanks, and grouped by parentheses to indicate tree structure. An item

can either be

i. A *syntactic type*, represented by a name to which a period has been appended,

ii. A *syntactic type with stated minimum multiplicity*, represented by a name with appended period, followed immediately by an integer.

iii. A *literal*, represented either by a quoted or unquoted character string; quotes may be omitted whenever a quoted string contains no characters which have special significance within a tree-describer part.

iv. A *set of literals*, enclosed within curly brackets, and separated by commas.

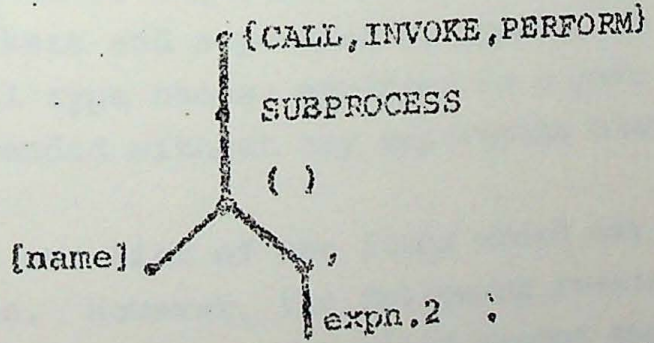
v. A *lexical type*, or *set of lexical types*, represented by lexical type names enclosed within square brackets and separated by commas where necessary.

Our list is still somewhat incomplete, but before we pile up additional cases a helpful illustration may be in order. Consider the somewhat hypothetical tree describer

(2) {CALL,INVOKE,PERFORM} (SUBPROCESS('( )'[name](, expn.2))

Since parenthetical grouping (by normal round parentheses) indicates tree structure, this tree description corresponds to the following binary tree:

(3)



Note in this example the general fact that in a sequence of items and parenthetic groups all at the same parenthesis level, one item (generally but not always the first) will determine

the type of a node, while the others will represent the descendants of this node. As the figure (3) makes plain, the tree descriptor (2) would correspond to a source-statement form beginning with one of three possible keyword sequences CALL SUBPROCESS, INVOKE SUBPROCESS, PERFORM SUBPROCESS, followed by the ordinary kind of function invocation consisting of a name (i.e., an item of lexical type 'name') followed by a parenthesized, comma-delimited list of expressions (i.e., compound items of syntactic type 'expn', two such items at least being required).

Having given this illustration, we go on with our list of item-types, next mentioning

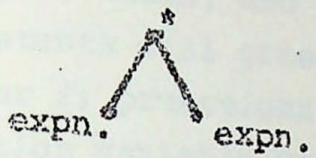
vi. *A lexical type or a set of lexical types with stated minimum multiplicity*, represented by a name or sequence of names enclosed within square brackets, to which a period and an integer are appended.

vii. A *literal of stated lexical type*, consisting of a quoted or unquoted string representing a literal, to which one or several lexical type names, enclosed in a pair of square brackets, are appended without any separating blank.

viii. A *set of literals of stated lexical type*, consisting of a group of quoted or unquoted character strings enclosed within curly brackets and separated by commas, to which one or several lexical type names, enclosed in a pair of square brackets, are appended without any separating blank.

This completes our list of the items which may appear in a tree-description. However, the following remains to be stated. In a sequence of items or parenthesized groups appearing in a tree description, all being at the same parenthesis level, and the length of the sequence being greater than one, at least one literal or set of literals must always appear. Such a sequence represents a non-terminal node and its immediate descendants. By convention, the first *literal* or *set of literals* appearing in this sequence determines the type of the node,

and the remaining parts of the sequence determine the descendants of the node, in order. This convention is useful in that, for example, it allows us to describe a tree of the form



either by the sequence

* expn. expn.

or by

expn. * expn. ;

the second is more apt to resemble the source form of a syntactic construction which we might be using the postparse metalanguage to specify.

It will be convenient, in the remaining parts of an alternative to refer to items in a tree description, *and to the relative parse tree address to which these items correspond*, by serial position in the tree description, counting from left to right and ignoring parentheses. Of course a group of bracketed items all correspond to the same tree address; the whole group is therefore counted only once. We append periods to integers used in this way as tree-location references; Thus, for example, the designation 1. , used in connection with the tree description (2) given above, refers to the topmost node of the associated tree (3); the designation 2. refers to 1.'s immediate descendant; the designation 5. refers to the intermediate node of type ',' and to its tree address etc.

Having established this convention, we now go on to describe the remaining parts of an alternative. An 'obligatory nonliterals' part (cf. (1) above) consists simply of a series of item references; the items designated should all be syntactic types rather than literals or lexicals. By their designation, they are noted as obligatory.

A pretest is a block of SETL code, standard in form, except that within it a node reference n. may be used as a macro for what would otherwise have to be written *partof(node,locator)*, where *locator* is the tree-address corresponding to the item reference (cf. also the text of the *partof* function, given above in the *postparse* code, p. 203). The same remark applies to pre-actions, post-tests, and postactions (cf. (1) above). Pretests and postests will generally set a communication variable *ok* either to t or f; preactions and postactions may set the value of a communication variable *val*. The whole block of code constituting a pretest, preaction, post-test, or post-action may be labeled with a name followed by a pair of colons; if this is the case, the pre-action may be invoked again simply by repeating this name, followed immediately by two periods.

A re-evaluation specification is a sequence of *units* separated by commas. Each unit may either be a quoted or unquoted literal string, an item-reference of the form

n.

where n is an integer, or a SETL code block assigning a value to the communication variable *val*. The first unit in the re-evaluation specification must be either a literal or a code block calculating a literal and assigning it to *val*; this literal will then be used to determine the node-type of the created node which is to be the root of the re-evaluated parse subtree desired. The remaining units specify, in sequence, the immediate descendants to be attached to this root, according to the following rules:

i. A literal string specifies a literal descendant (a lexical type must be specified for every literal used in this way).

ii. An item reference determines a tree address or *locator*, and serves to designate *value(partof(node,locator))* as the quantity to be attached to the re-evaluated subtree.

However, an item reference to a multiple item locates a parent node, and then directs the attachment of a whole sequence *value(desc(partof(node,locator),))* to the re-evaluated subtree.

iii. A code block will construct some treelet of acceptable form, and set the communication variable *val* equal to the root of this treelet, which will then be attached to the re-evaluated subtree which is being constructed.

Since the terminating periods of successive item references can serve as separators, commas may be elided when such a sequence occurs within a re-evaluation specification.

The above remarks apply to *composite* alternatives, i.e. to those whose tree-description part contains more than one item. *Elementary* alternatives, whose tree description part consists of one single item, obey conventions slightly different, which are as follows.

i. Only one such alternative may contain a syntactic type name. This alternative then designates the *successor type* of a syntactic type, and should not include any other information.

ii. All other elementary alternatives will consist either of single literals, a single group of literals, single lexical type names, or of a single group of lexical type names. Such alternatives should omit the name-specification part as the evidential-weight part ordinarily present.

The mapping *altset(syntype,nodtype)* used (cf. p. 196, item 19, also items 20 and 21) to ensure efficiency in locating those alternatives of a syntactic type which it is appropriate to consider in a given context will be built up as follows. The set of all the characters which occur as the node types of the top node of any composite alternative of the syntactic type will be collected, and, for each of these, the set of all alternatives in which it occurs will be formed. This set will be the value of *altset(syntype, nodtype)*. If we exclude the one exceptional alternative which may be used to define the successor type of a syntactic type, single item alternatives must either be lexical types or literals; all such will be collected into the set *lexalts(syntype)*.

Occasionally however sets of composite alternatives formed in this way will be inappropriately large. When this is the case, one can proceed as follows: take all the alternatives which belong to such a large group and which have a node of given type t as their root. Considering them all together, find some particular immediate-root-descendant position, say the j-th, present in all of them, and such that in this position there will occur characteristic literals allowing the large group of alternatives which concerns us to be broken into smaller groups. The indicate that all alternatives whose root node is of type t are to be resolved by reference to the j-th position; this is done merely by adding the resolution specification

/rs/j

to one such alternative. This will define j as a value of the function lookey(syntype, nodtype) (cf. p. 126, item 20). The literals present in the j-th position in any alternative having root type t will then be collected, and this information used to define an appropriate subcollection *secaltset(synt,liter* (cf. p. 126, item 21) which hopefully will consist of just a few alternatives.

The whole body of a tree grammar should be prefixed by a string of the form

/th/integer/

where the integer which appears defines the *guessing threshold* to be used, cf. p. 195, item 13. The grammar should be terminated by an appearance of

/end/ .

The *root syntactic type* of the whole grammar, which defines the value which the argument *syntype* will have when *postparse* is first called, is by definition the first syntactic type named in the grammar.

After these rather dry descriptions of facilities, it will come as a relief to begin to use the facilities depicted at such great length in the preceding pages. We now give a complete definition, in the postparse metalanguage, of the 'generalized BASIC' which has already served us several times as an example. The lexical and the precedence structure of this language have already been defined; naturally enough, our postparse description of the language will refer to some of the conventions established in these earlier definitions, which should be consulted as necessary; cf. pp. 122-125; and 178-181. It is worth pointing out that the descriptive methods which we have developed allow diagnosing parsers to be described in a very efficient way; the total mass of material needed to define the entire 'front end' of a generalized BASIC compiler amounting to not much more than three pages.

Here is the required definition, written in the postparse metalanguage.

Description of the generalized BASIC language in the postparse metalanguage.

```
/th/10//
labstat = [nm]: stat./labeled statement.keyword./10
        = stat. //
stat    = let(var.=exp.)/statement.keyword.assignment statement
                /10,5/rv/=,2.3.
        = read(, var.2)/read statement/10,0/rv/read,3.
        = read var. / read statement/10
        = print(, expn.2)/print statement/10,0/rv/print,3.
        = print expn./print statement/10
        = data(, numb.2)/data statement/10,0/rv/data,3.
        = data numb./data statement/10
        = goto[nm]/goto statement/10
        = go(to[nm])/goto statement/8,3/rv/goto,3.
        |= if((expn.{>,<}{>,=}expn.)) then [nm])/if statement/
                5,5,3,5
           /tl/ok=n((ntl is nodtype(3.)) eq '>'
           and(nt2  is nodtype(4.)) eq '>';
            ,contains an illegally structured comparison operat
           /rv/val={<'>','=','ifge'>,<'<','>','ifne'>,
                    <'<','=','ifle'>}(ntl,nt2);,2.5.7.
        =]if((expn.{>,<,=} expn.) then [nm])/if statement/5,3,5
           /rv/val={<'<','iflt'>, <'>','ifgt'>,
                    <'=','ifeq'>} nodtype(3.);,2.4.6.
        = for(([nm]=expn.) to expn.)/for statement/6,1,3
           /rv/forst1,2.4.6
        = for((([nm]=expn.) to expn.) step expn.)/for statement/
                6,1,3,6 /rv/forst2,2.4.6.8
        = next[nm]/next statement/10
        = dim(,dimelt.2)/dimension statement/10,0/rv/dim,3.
        = dim dimelt./dimension statement/10
        = gosub[nm]/call statement/10
        = return omitted[var]/return statement/6,6/rv/return[var]
```

```
        = def(var.=exp.)/function definition/8,2/rv/def,2.4.
        = end omitted[var]/end statement/6,6/a2/end=t;
        = blankstatement[var]/blank record/10 //
expn    = + expn.2/expression.operator or some illegal token
          used to represent a quantity.sum/10
        = + expn./positive quantity/10
        = - expn.2/difference/10
        = - expn./negative quantity/10
        = / expn.2/quotient/10
        = '.exp.' expn.2/exponential/10
        = '( )' [nm] {, expn.2)/indexed expression or function
          call/8,4  /rv/apply,2.4.
        = '( )'[nm]expn. / indexed variable or function call/0
          /rv/apply,2.3.
        = '( )' expn. /parenthesized expression/10/rv/2.
        = [irh, ir, dip, r, nm] //

var     = '( )' [nm] {, expn.2)/indexed expression or function
          symbol. comma, parenthesis, or other token.
          /8,4/rv/store,2.4
        = '( )' [nm] expn./simply indexed expression or functior
          symbol/10/rv/store,2.3.
        = [nm]                          //
numb    = - [irh,ir,dip,r]/constant.number sign. negative
          number/10
        = [irh,ir,dip,r]                //
dimelt  = '( )' [nm] (,[irh,ir].2)/dimension element.part.
          /8,4  /t2/ ok= #desc(.3) le 3; , is legally indexed
          beyond three dimensions  /rv/ dimension 2.4.
        = '( )' [nm] [irh,ir]/dimension element/10
          /rv/dimension,2.3.    //
/end/
```

...The parsing method which has been described above in so much detail permits of a highly efficient and compact machine-level implementation, covers a wide class of practical programming languages, and is of easy use. Of course, many other parsing methods exist. We shall go on to describe certain of these methods, and certain interesting algorithms associated with them, in more detail. One of the most interesting of parsing techniques is the 'nodal span' method of Cocke, Younger, and Earley; a detailed discussion of this algorithm will be found in Cocke and Schwartz, *Programming Languages and Their Compilers*, second revised version, New York University, April 1970, section 4.6. In contrast to most other parsing procedures, this method is highly stable in the presence of ambiguity; in particular, it can be applied to any context-free grammar whatsoever; moreover, the method can readily be generalized to apply to grammars more flexible than simple Backus grammars. It can also be adapted to grammars in which a central group of exactly correct rules is surrounded by a penumbra of partly incorrect rules, and in this adaptation can be used to assign a quantitative 'degree of grammaticalness' to sentences which are not quite grammatical. It also suggests interesting methods for the parsing of structures, such as plane figures, which are more general than linear strings.

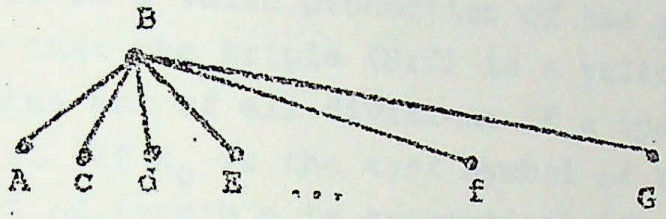
In describing this algorithm, the notion of a *context free* or *Backus* grammar is convenient. Such a grammar is defined by giving

i. An alphabet of tokens, called the *alphabet* of the grammar; the symbols in this alphabet are distinguished into two classes, *intermediate* symbols and *terminal* symbols. We shall often indicate intermediate symbols by capital letters, terminal symbols by small.

ii. For each intermediate symbol B, a finite set of *productions* of the form

(1) $$B \rightarrow AcdE...fG \ ,$$

where the right-hand side of a production may be any finite sequence of intermediate and terminal symbols. Heuristically, the inclusion of the production (1) in a grammar means that the symbol B, whenever it appears in a sentence of the language which the grammar describes, can be replaced by a string AcdE...fG . Conversely, this means that it is possible to cover the string AcdE...fG by a treelet



when sentences are being supplied with parse trees.

iii. One intermediate symbol of the grammar must be designated as its *root* or *sentence* type. Valid sentence-forms are then all those strings of symbols which can be produced from the root symbol by the successive application of productions of the grammar; valid sentences are all those valid sentence-forms containing terminal symbols only.

Having said this, we prepare to present a SETL program for the nodal span parsing algorithm by giving a brief review of this algorithm's structure. The algorithm accepts an input sequence of tokens and a context-free grammar, and parses the tokens according to the grammar. The grammar may be ambiguous or unambiguous, but, in order to simplify our exposition, we take it to be given in a standardized "reduced" form in which each production of the grammar either has the form A→a or the form A→BC, A, B, and C denoting intermediate symbols of the grammar, and a denoting a terminal symbol of the grammar. In this situation we call a triple (pAq) consisting of two integers p, q and an intermediate grammatical symbol A a *span*. We say that this span (pAq) is *present* in the input if the string of tokens extending from the p-th through the q-1'st position in the input string can be generated from the symbol A using

productions of the grammar. For q = p+1 this will clearly be the case if and only if some production A→a generates the p-th character of the input string; for q > p+1 this will equally clearly be the case if and only if there exists some r, p < r < q, and two intermediate symbols B, C of the grammar such that both spans (pBr) and rCq) are present in the input, and such that A→BC is a valid production of the grammar. In this case we say that the triple (BrC) is a valid *division* of the span (pAq); the set of all divisions of a span is called its *division list*. If A_0 is the *root* symbol of the grammar an input string I of length n is grammatical if and only if the *root span* $(1A_0(n+1))$ is present in it. In this case, producing the collection of all division lists for all spans present in I may be regarded as equivalent to parsing I; if these lists are available the ordinary "parse tree" of I can be obtained in a perfectly direct way by dividing the basic span into parts in all the ways indicated by its division list, dividing all these parts into sub-parts using their division lists, etc. until spans of unit length are reached.

It is also clear that not all spans present in the input, but only those spans which are obtained from the basic span by this process of division, are *relevant* to the final analysis of the input. Other spans present in the input are in effect merely false starts never resulting in a complete parse. The set of division lists belonging to the narrower collection of spans relevant in this sense is called the *cleaned* division list for the parse of the input string. The above remarks should make clear the structure of our parsing algorithm. We proceed from left to right, accumulating spans present in the input string. On encountering the n-th symbol a of the input string, we add a span (nC(n+1)) to our collection whenever the existence of a production C→a justifies this. Newly added spans (rCq) are then "processed" by locating all spans (pBr) for which the existence of a production A→BC in the grammar will yield a new span (pAq). Each time two

spans are combined in this way to give a span (pAq) we make an appropriate entry on the division list of the span (pAq). When as many spans as possible have been produced by this combination process, we go on to consider the next input token in turn, and so forth to the very end of the string of input tokens. When the end of the string of input tokens has been reached we check to see whether or not the basic span $(1A_0(n+1))$ is present; if this basic span is present, we prepare a cleaned set of division lists by the division process described above.

The SETL program given below carries out the steps just outlined, determining at the same time whether the given input has an ambiguous or unambiguous parse; the criterion for ambiguity is simply that at least one of the division lists in the cleaned set should contain more than one element. A few additional preliminary remarks will cast light on the details of the program which follows. Spans are represented as ordered triples <q,A,p> composed of two integers p, q and an intermediate grammatical symbol A. For technical reasons the larger integer q is placed first and the smaller second. The family of division lists belonging to such spans is maintained as a collection of quadruples <<q,A,p>, r,B,C> associating with each span all its divisions (BrC). The context-free grammar according to which the input is to be parsed is taken in the program which follows to be the collection of all triples <B,C,A> corresponding to productions A→BC of the grammar. A function *syntypes*, which maps each character a of the input string onto the set of all intermediate symbols A for which there exists a production A→a, is also presumed in the following program.

Here then is a SETL version of the basic algorithm for parsing by the method of nodal spans.

```
define nodparse (input,gram,root,syntypes,spans,divlis,amb);
todo = nl; divlis=nl; spans={<2,s,1>, s∈syntypes(input(1))};
(1< ∀n ≤ #input)
todo= {<n+1,s,n>, s∈syntypes(input(n))};
spans = spans ∪ todo;
(while todo ne nl) next from todo; do makenewtodofrom(next);
end while; end ∀n; /*check on grammaticalness*/
if n (topspan is <#input+1,root,1>) ∈ spans then
  <spans,divlis,amb>=<nl,nl,f>; return;;
/* else clean up set of spans and determine ambiguity */
spans = nl; amb=f; getdescs(topspan); /*clean division list*/
divlis={d ∈ divlis | hd d ∈ spans}; return;
/*auxiliary block */ block makenewtodofrom(next);
<end,typ2,mid>  = next;
(∀spend ∈ spans{mid}, type ∈ gram{typ1 is hd spend,typ2})
newsp=<end,type,tl spend>; <newsp,mid,typ1,typ2> in div;
if n newsp ∈ spans then newsp in spans; newsp in todo;;
end makenewtodofrom; end nodeparse;
define getdescs(top); /*auxiliary tree-walker */
nodparse  external divlis, spans, amb;
if top ∈ spans then return;; top in spans;
if (#divlis{top}) gt 1 then amb=t;; <end,-,start>= top;
(∀x ∈ divlis{top}) <mid,typ1,typ2> = x;
getdescs(<end,typ2,mid>); getdescs(<mid,typ1,start>);;
return; end getdescs;
```

In unfavorable cases, the nodal span parsing method may generate reasonably large numbers of spans not relevant to the final analysis of an input string. J. Earley has introduced an interesting modification of nodal span parsing which improves its performance in this regard. In Earley's method, one is always sure, as one scans from left to right over an input string $S = t_1\ t_2 \ldots t_m$ of tokens, that a span pA_{q+1} will never be formed unless there is some continuation $t_1 \ldots t_q\ t'_{q+1}\ t'_{q+2} \ldots$ of the initial portion $t_1\ t_2 \ldots t_q$ of S

to whose analysis pAq+1 is relevant. Thus one never forms spans which are not 'relevant to as much of the input as has been scanned.'

The method is this. Immediately before scanning the n-th token of S, one calculates a set, called *startat(n)* in the algorithm which follows, which consists of all those intermediate symbols of the grammar Γ being considered which can validly follow the part of S already scanned. The rule for forming this set is simple and iterative: If there exists a span pBn, and intermediate symbols A and C such that A→BC is a production of Γ and such that A ε *startat(p)* , then any symbol D for which there exists a string DEF... derivable from C by productions of Γ belongs in *startat(n)* . Having constructed these sets, we use them in the remainder of the algorithm to control the formation of additional spans, always applying the following test: no span pAq is to be formed for which A ε *startat(p)* is false.

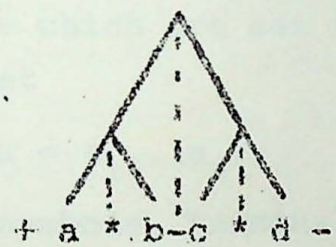
This improvement may be particularly significant if the nodal span parsing method is to be applied to the analysis of natural language, as in natural languages single words tend to be highly ambiguous as to syntactic type, and restrictions on the syntactic classes which can appear at a given point may therefore be most helpful.

In other respects, Earley's procedure, for which we now give a SETL algorithm, is very close to the nodal span parsing algorithm set forth above.

```
/*note use below of closure routines given on p. 132*/
define earleyparse(input,gram,root,syntypes,spans,divlis,amb);
/* first calculate the fixed relationships needed*/
begins = {<<*z3>x, hd x , x∈gram};
descends = begins u {<<*z3>x,<*z2>x> ,x∈gram};
symbs =   close (descends,{root}); ultbegin=closef(begins,symbs);
/* the above need not be repeated unless the grammar is changed*/
todo=nl; divlis=nl; startat={<1,ultbegin{root} >}; spans={<2,s,1>,
  s∈syntypes(input(1))|s∈startat(1) }; (1 < ∀n ≤ #input)
/* calculate startat(n) */ pops=nl; (∀spanend∈spans(n))
<typ1,beg> = spanend;
pops=pops u {hd x, x∈gram{typ1}|                    (tl x) ε startat(beg)};;
startat(n) = ultbegin {pops};
/*now go on to form spans */
todo = {<n+1,s,n>, s∈syntypes(input(n))|s∈startat(n)};
spans = spans u todo;
(while todo ne nl) next from todo; do makenewtodofrom(next);
end while; end ∀n; /* check on grammaticalness */
if n (topspan is <#input+1,root,1>) ε spans then
  <spans,divlis,amb> = <nl, nl,f> ; return;;
/*else clean up set of spans and determine ambiguity */
spans = nl; amb = f; getdescs(topspan); /*clean division list*/
divlis = {d ε divlis | hd d ε spans}; return;
/* auxiliary block */ block  makenewtodofrom(next);
<end,typ2,mid>= next;
(∀spend ε spans{mid}, type∈gram{typ1 is hd spend,typ2}
                      |type ε startat(tl spend)}
  newsp=<end,type,tl spend>;
<newsp,mid,typ1,typ2> in div; if n newsp ε spans
  then newsp in spans; newsp in todo;; end makenewtodofrom;
  end earleyparse;
/*a copy of the subroutine 'getdescs' occurring on page 223
  is also required here */
```

The next parsing method which we shall describe, the so-called *generalized precedence method* of McKeeman and others, elaborates extensively upon the basic idea used in the very simple precedence preparser described in the first pages of the present section, and, having done so, finds itself able to dispense with any separate postparse. The method used is interesting, and can yield rather fast parsers; it has however a number of disadvantages, being, among other things, somewhat unapt for the production of diagnostics. A bottom-up method complains when asked to do work that is naturally top-down!

The key idea of generalized precedence parsing is as follows. Parsing ascribes a tree to a string of tokens; more precisely, it attaches the tokens of an input string to the twigs of a parse tree. For example, we may represent the parse of an expression ...+a*b-c*d-... as follows



If a token (such as a or c in our example) is attached to a left-hand twig of the parse tree, we say that the parse *leans right* from the token; if a token (such as b or d in our example) is attached to a right-hand twig, we say that the parse *leans left* from the token. It is intuitively clear that if we can always tell which way the parse leans at each token, we can reconstruct the entire parse tree, using the following method: Scan the input string, and find the first token t at which the parse leans left. Then find the last token t' preceding t at which the parse leans right. The string of tokens between t' and t must all be descendants of some single node in the parse tree. Building a treelet to which this string of tokens is attached, and replacing or so to speak *condensing* this string of tokens into a single token representing the root of this treelet, we obtain a shorter string, to which the same

process can be applied again. Repeated condensations will eventually reduce the whole of a token string to a single symbol; in the process, its parse tree will have been built.

Next observe that we can generally expect to be able to deduce the direction in which the parse tree of a string leans from a given token t by examining a very few tokens in the immediate neighborhood of t. This is the basis for the simple precedence parse considered earlier; the tree always leans in the direction of that nearby operator sign having highest precedence. Even for languages too complex for this ultra-simple rule to work infallibly, a slightly more sophisticated variant of the same method may be feasible; we call languages for which this is the case *extended precedence languages*.

In the logical setting defined by a context-free grammar Γ, we may give formal definitions capturing the ideas concerning extended precedence which are set forth above. These definitions are as follows. Let

$$\alpha = a_1 \ldots a_n \; , \qquad \beta = b_1, \ldots, b_m \; , \tag{2}$$

be two strings of symbols, terminal or non-terminal, of the alphabet of Γ. We define three relationships, $\alpha \mathrel{\cdot\!>} \beta$, $\alpha \doteq \beta$ $\alpha \mathrel{<\!\cdot} \beta$ between such strings. The definitions are as follows:

i. $\alpha \mathrel{\cdot\!>} \beta$ (heuristically: the parse must lean toward α from the last symbol of α) if $\ldots a_1 \ldots a_n \; b_1 \ldots b_m \ldots$ can occur within a sentence σ, in whose parse tree a_n is attached to the rightmost twig of some subtree.

ii. $\alpha \mathrel{<\!\cdot} \beta$ (heuristically: the parse must lean toward β from the first symbol of β) if $\ldots a_1 \ldots a_n b_1 \ldots b_m \ldots$ can occur within a sentence σ, in whose parse tree b_1 is attached to the leftmost twig of some subtree, while a_n is not attached to the rightmost twig of a subtree.

iii. $\alpha \doteq \beta$ (heuristically: the parse is balanced at the last symbol of α) if $\ldots a_1 \ldots a_n b_1 \ldots b_m$ can occur within a sentence σ, in whose parse tree a_n and b_1 are both attached to

twigs with a common parent node.

If at most one of these three relationships holds between α and β we call the pair α,β an *unambiguous context*. (If none holds, we call α,β an *impossible context*; it is easy to see that in this case $\ldots a_1 \ldots a_n b_1 \ldots b_m \ldots$ can never occur within a grammatical sentence.) If the language defined by Γ is such that whenever α is a sequence of symbols of length n and β is a sequence of symbols of length m it follows that α,β is an unambiguous context, then we say that Γ is an *n,m-generalized precedence grammar*. In this case, the generalized precedence parsing method outlined above can be used to reconstruct the parse tree of any sentence valid according to Γ.

Parsing by this scheme will plainly involve the use of tables which state the relationships which hold between sequences α and β of symbols. We shall now present a SETL algorithm which, given a grammar (and given limits n,m for the lengths of left- and right-hand contexts to be considered) will produce such tables. The basic idea of the algorithm is quite simple. We start with a pair of single symbols, and determine whether or not these 1-token strings constitute an unambiguous context. If they do not, we add one symbol of additional context, first on the left, then on the right, looking always for unambiguous contexts. This will generate a sequence of tables which collectively contain the required precedence information. If when we have reached the maximum length-limit for contexts to be considered ambiguities still remain, then it follows that the grammar being considered is not an n,m-generalized precedence grammar.

The algorithm which follows immediately below is a quite straightforward elaboration of this idea. It works with left- and right-hand contexts of the lengths (1,1), (2,1), (1,2), (2,2), (3,2), (2,3), (3,3) in sequence, up to a stated maximum length. A data-structure *table(seqa,seqb)* which is progressively built up records values 1, 2, and 3, depending on the relation

of precedence that holds between the sequences seqa and seqb of characters; the value © is used if this relationship is ambiguous. Ambiguous pairs are held temporarily in a set *ambig*.

An output flag *definite* is set to t if the situation analyzed has the *lmax,rmax* precedence property; to f otherwise. The definitions of the basic relationships *seqa* <> *seqb* , etc. given above convert very directly into algorithmic form. A few simple auxiliary processes for concatenating sequences, selecting parts of sequences, etc. are used.

The algorithm which follows assumes that a context-free grammar *gram* and its alphabet *chars* are both given. The grammar *gram* is assumed to be a set of pairs <*int*,*seq*>, where *int* is an intermediate symbol, and *seq* is the sequence of characters corresponding to the right-hand side of a grammatical production. In what follows, a function *isgram*(x), which determines whether the sequence x is part of a grammatical sentence, is also used. The SETL code for this function will be given somewhat below, after we have discussed some auxiliary algorithms for the transformation of context-free grammars into standard form.

```
definef mckeetable(lmax,rmax,gram,chars,definite);
/* first calculate fixed tables useful below*/
origins={<tl x, hd x>, xegram};
/* next get initial parts of right-hand sides */
leftcovers = {<k sfirst tl x, hd x>, xegram, 1 ≤ k ≤ #tl x};
rightcovers = {<k slast tl x, hd x>, xegram, 1 ≤ k ≤ #tl x};
covers = {<k sfirst (l slast tl x), hd x>, x ε gram,
          1 ≤ l ≤ #tl x , 1 ≤ k ≤l};
/* initialize */ ambig=chars; <m,n>=<1,0>; table=nl;
          (while m le lmax or n le rmax)
leftright = if m gt n then 0 else 1;
if m gt n and n gt 0 then m=m-1;;
if m le lmax and n le rmax then do extendres(m,n,leftright);;
  end while;
if ambig ne nl then print 'ambiguous cases remaining:', ambig;
definite = f;
else definite = t; end if; return table;
block extendres(m,n,leftright);
/* extends context one symbol to left or right;
  left chosen if leftright equals 1, otherwise right */
m = m + leftright; n = n+(1-leftright); oldamb=ambig;ambig=nl;
(Wampairεoldamb, cεchars) <seqa,seqb>=ampair;
/*extend by 1 character to right or left */
if leftright eq 1 then seqa = {<1,c>} splus seqa;
  else seqb=seqb splus {<1,c>};;
poss = nl; /* set of relations which are possible */
if ∃xεrightcovers(seqa)|isgram({<1,x>} splus seqb)  then 1 in poss;;
if 1 ≤ ∃n < #seqa, xεorigin(n slast seqa)|
  isgram(#seqa-n sfirst seq splus {<1,x>} splus seqb)
          then 1 in poss;;
if seqa evenwith seqb then 2 in poss;;
if ∃xεleftcovers(seqb)| seqa evenwith {<1,x>) then 3 in poss;;
```

```
if 1 ≤ ∃n < #seqb, xcorigin{n sfirst seqb}|
  seqa evenwith ({<1,x>} splus (#seqa - n slast seqb))
        then 3 in poss;;
/* now set up table entry or classify as ambiguous */
if #poss eq 1 then table(seqa,seqb) = ↓poss;
  else <seqa,seqb> in ambig; table(seqa,seqb)=0;;
end ∀ampair; end extendres; end mckeetables;
definef seqa evenwith seqb;
mckeetable external covers, leftcovers, rightcovers,origins;
/* auxiliary routine to test 'equality' case of precedence */
return if covers{seqa splus seqb} ne nl then t
  else if 1 ≤ ∃k < #seqb, xcrightcovers{seqa splus
        {k sfirst seqb}}
  |isgram{{<1,x>} splus {#seqb-k slast seqb}} then t
    else if 1 ≤ ∃k < #seqa, xcleftcovers{k slast seqa splus seqb}
  |isgram{#seqa-k sfirst seqa splus {<1,x>}} then t
    else if 1 ≤ ∃k < #seqa, 1 ≤ ∃l ≤ #seqb,
        xcorigins{k slast seqa splus {l sfirst seqb}}
  |isgram{#seqa-k sfirst seqa splus {<1,x>} splus
        (#seqb-l slast seqb)} then t
  else f; end evenwith;
/* now some short auxiliary routines for handling sequences */
definef seqa splus seqb; return seqa ∪ {<hd x+#seqa,tl x>,xeseqb};
  end splus;
define n sfirst seq; return {<k,seq(k)>, 1≤k≤n}; end sfirst;
define n slast seq; return {<k,seq(k+#seq-n)>, 1≤k≤n};
  end slast;
```

Next we discuss the manner in which a grammar of the form considered in connection with the preceding algorithm can be transformed into one of the more restrictive 'reduced' form considered in connection with the nodal span parsing method (cf. p. 223). Observe that in the following discussion we mean to consider two grammars as equivalent if their alphabets contain the same terminal symbols and if they generate the same set of valid sentences, i.e., non-empty valid strings containing terminal symbols only. Suppose then that a grammar is given as a set of productions of the general form

(1) $$B \rightarrow AcdE...fG$$

We will, to start with, admit productions having null right-hand sides (these might be called 'erasing' productions) and our first aim will be to see how a grammar Γ containing such productions can be transformed into one which contains no such productions. This transformation is easy enough, and may be accomplished as follows.

i. Call an intermediate symbol B *erasable* if the null-string can be produced from it by some sequence of productions. Note that B is erasable if either B is the left-hand side of some erasing production

$$B \rightarrow$$

or if there exists a production

$$B \rightarrow CDE...F$$

such that every symbol occurring on the right-hand side is known to be erasable.

ii. Drop all erasing productions from Γ. At the same time, given any production (1) of Γ, introduce as additional productions of Γ all those productions which result from (1) by dropping some set of erasable symbols from its right-hand side, but in such a way that at least one symbol remains on the right-hand side.

The set of productions obtained in this way constitutes a new grammar, equivalent to Γ, but containing no erasing productions.

Assuming that as in our last algorithm a grammar is given as a set of pairs <*lnt*, *seq*> where *seq* is a sequence of characters, we may express this simple procedure in SETL as follows.

```
erasables = {hd x, xεgram | tl x eq nl};
(while ∃[x]εgram|n hd xεerasables and erasables incs tl[tl x])
  hd x in erasables;;
newgram=nl; (∀xεgram) <symb,seq>=x;
  eraselocs={1≤n≤#seq|seq(n)εerasables};
(∀spotsetεpow(eraselocs)) newseq=nl; (1≤n≤#seq)
if n n ε spotset then newseq(#newseq+1)=seq(n);; end ∀n;
/* now have built up new right hand side */
if newseq ne nl then <symb,newseq> in newgram;; end ∀spotset;
  end ∀x;
/* replace old grammar by new */ gram=newgram;
```

Next we aim to eliminate all single-character productions

(2) $$A \to B$$

of one intermediate symbol from another, and, at the same time, to eliminate terminal symbols from the right-hand side of every production except those of the single-symbol form

$$A \to b \quad .$$

This again is an easy transformation, and can be accomplished as follows:

iii. For each terminal symbol a, introduce a new intermediate symbol $\tilde{A}$. Replace each occurrence of a on the right-hand side of a production of Γ by an occurrence of $\tilde{A}$. Then add the productions

(3) $$\tilde{A} \to a$$

to Γ.

iv. Call A a *prime ancestor* of B if the single symbol B can be produced from A by some sequence of productions of the grammar Γ. Drop all single-intermediate-symbol productions (2) from Γ. At the same time, wherever there exists a production

$$B \rightarrow CDE\ldots$$

with more than one symbol on its right-hand side, introduce a new production

$$A \rightarrow CDE\ldots .$$

In SETL, this procedure may be written as follows (note that, as in the McKeeman table algorithm, *chars* denotes the alphabet of our grammar).

```
/* first determine intermediate symbols */ inter=hd{gram};
term={x∈chars|n x∈inter}; intof=nl; newg=nl;
(∀t ∈ term) intof(t) = newat;;
(∀x ∈ gram)<symb, seq>= x;
<symb,{<n,if(int is intof(seq(n)) ne Ω then int else seq(n)>,
        1<n<#seq}> in newg; end ∀x;
/* which accomplishes step iii of the text */
singles = {<tl ∋ tl x, hd x>, x∈newg|(#tl x) eq 1};
/* determine 'prime ancestor'    relationship by closure,
        cf p. 132 */
pransc= closef(singles, inter); newgram=nl;
(∀x ∈ newg|#tl x gt 1)<symb,seq>= x; x in newgram;
(∀y ∈ pransc{symb}) <y,seq> in newgram;; end ∀x;
/* now add terminal single-symbol productions to grammar,
  and set grammar to new value */
gram = newgram u {<intof(t), {<1,t>}>, t∈term};
```

As transformed thus far, the grammar contains productions of two types; productions A→a, and productions A→BCD...EF, where at least two symbols appear on the right, and where all symbols appearing on the right are non-terminal. For every production of the second kind whose right-hand side contains more than two symbols, we introduce a sequence of auxiliary intermediate symbols G, G', ..., and replace the production A→BCD...EF by a set of productions

$$A \rightarrow BG$$
$$G \rightarrow CG'$$
$$G' \rightarrow DG''$$
$$\cdots$$
$$G^{(n)} \rightarrow EF$$

This step, which essentially completes the transformation of our grammar into the desired form, may be written in SETL as follows.

```
newgram = {xegram | (#tl x) le 2};
(∀xegram|(#tl x) gt 2)<left,seq>=x;
(1 ≤ ∀n ≤ #seq-2) aux=newat; <left,{<1,seq(n)>,<2,aux>}>in newgram;
left = aux; end ∀n; <left,{<1,seq(#seq-1)>,<2,seq(#seq)>}>
  in newgram;
end ∀x; gram=newgram;
```

A grammar in the precise form required by the nodal span parsing algorithm is then related to a grammar in the reduced form we have attained by making the following trivial final transformation.

```
newgram = {<(tl x)(1), (tl x)(2), hd x>, xegram|(#tl x) eq 2};
synt = {<(tl x)(1), hd x>, xegram|(#tl x) eq 1};
syntypes = {<x, synt{x}>, xehd[synt]};
```

The transformations described in the last few pages enable us to specify the *isgram* function required by the McKeeman table algorithm given above. The main difficulty to be overcome lies in the fact that, given a string S of tokens, we wish to determine if S *is a part of* any grammatical sentence form. Our method will be to extend the grammar Γ with which we deal, obtaining a larger grammar Γ' which generates all strings S which are parts of strings generated by Γ. This we do as follows. For each intermediate symbol A of Γ, introduce three additional symbols, which we call A_ℓ, A_r, and $A_{\ell r}$; if a is a terminal symbol, a_ℓ, a_r, and $a_{\ell r}$ will all be equal to a. For each production

(4) $$A \to BCD \ldots EFG$$

of Γ, introduce new productions

$$A_\ell \to B_\ell CD \ldots EFG$$

$$A_\ell \to C_\ell D \ldots EFG$$

$$A_\ell \to D_\ell \ldots EFG \quad , \quad \text{etc.}$$

by removing zero or more symbols from the left-hand end of the right side of (4). Similarly, introduce productions

$$A_r \to BCD \ldots EF_r$$

$$A_r \to BCD \ldots E_r \quad , \quad \text{etc.}$$

by removing zero or more symbols from the right-hand end of the right side of (4); and productions

$$A_{\ell r} \to C_\ell D \ldots E_r \quad , \quad \text{etc.}$$

by removing zero or more symbols both from the right and from the left-hand and of the right of (4).

Next, take the root symbol $\bar{A}$ of Γ, introduce a new root symbol $\hat{A}$ for Γ', and introduce three productions,

$$\hat{A} \to \bar{A}_\ell$$

$$\hat{A} \to \bar{A}_r$$

$$\hat{A} \to \bar{A}_{\ell r} .$$

For each symbol D which occurs on the right-hand side of a production (4), introduce a production

$$A_{\ell r} \rightarrow D_{\ell r} .$$

Finally, introduce productions

$$A_{\ell} \rightarrow A$$

$$A_{r} \rightarrow A$$

for each intermediate symbol A. The grammar Γ' we require consists of all these productions, and has $\hat{A}$ as its root symbol; we leave it to the reader to show that Γ' generates precisely those sentence-forms which are parts of sentence-forms generated by Γ.

Assuming that *root* designates the root-symbol of Γ, the construction of Γ' from Γ may be depicted in SETL as follows.

```
inter= hd{gram};  ℓ=nℓ; r=nℓ; ℓr=nℓ;
(∀x∈inter) ℓ(x)=newat; r(x)=newat, ℓr(x)=newat;;
newroot=newat; extgram=gram; /*initialize new grammar */
(∀x∈inter)<ℓ(x),{<1,x>}> in extgram;<r(x),{<1,x>}> in extgram;
(∀seq∈gram{x},1≤n≤#seq)<ℓr(x),{<1,ℓr(seq(n))>}> in extgram;
<r(x),{<m, if m lt n then seq(m) else r(seq(m))>,
          1 ≤ m ≤ n}> in extgram;
<ℓ(x),{<m-n+1, if m gt n then seq(m) else ℓ(seq(m))>,
          n ≤ m ≤ #seq}> in extgram;
(1 ≤ ∀nn < n) <ℓr(x),{<m-nn+1, if m eq nn then ℓ(seq(m))
          else if m eq n then r(seq(m)) else seq(m)>;
          nn ≤ m ≤ n} in extgram;;
end ∀seq; end ∀x; extgram=extgram with<newroot,{<1,ℓ(root)>}>
                                  with<newroot,{<1,r(root)>}>
                                  with<newroot,{<1,ℓr(root)}>;
```

Having defined *extgram* in this way, we transform it using the processes described in the last few pages, to obtain a grammar *newgram* and a mapping *syntypes* of the forms required by the nodal span parsing algorithm. The required function

isgram(string) may now be expressed simply as follows.

```
nodparse (string, newgram, newroot, syntypes, spans, divlis, amb);
return /* we assume this occurs within a function */ spans ne nl;
```

8. Algorithms for permutations.

A *permutation* of a set s of n objects is a one-to-one mapping of s into itself. The set of permutations of s forms a group, indeed, the prototype of all groups; the combination and inverse being given as follows:

```
definef f c g; return {<x,g(f(x))>, x ε hd[f]}; end c;

definef inv f; return {<tl x, hd x>, x ε f}; end inv;
```

Permutations can conveniently be represented in the so-called *cycle form*. The cycle form of a permutation f of s is a family of disjoint sequences, collectively covering s, such that f(x) is always the next element in sequence after x, unless x is the last element of a sequence, in which case f(x) is the first element of the same sequence. A SETL algorithm for putting a permutation into cycle form may then be written as follows:

```
definef cycform(f); s=hd[f]; cycs=nl;
(while s ne nl) elt from s; cyc={<1,elt>}
(while(e is f(elt)) ε s doing elt=f(elt);) cyc(#cyc+1)=e;
        s = s less e;;
cyc in cycs; end while s; return cycs; end cycform;
```

Since the cycle form of a permutation represents the permutation in a condensed and structurally revealing manner, it is useful to be able to perform the basic operations of combination and inversion directly on the cycle form of a permutation. Inversion is easy; we merely reverse every cycle. In SETL this is:

```
definef invc cycs; inv=nl; (∀c∈cycs){<n,c(#c-n+1)>,1≤n≤#c}in inv;
  return inv; end invc;
```

The existence of the cycle form of a permutation shows that any permutation may be written as a product of 'cyclic' permutations, i.e., permutations of the special form a→b, b→c, ..., d→e, e→a. (The notation (abc...de) is often used for a permutation of this special sort.) A number of

algorithms for the inversion of permutations ('in place', i.e., without the use of extra storage space, assuming that the permutation is represented by a permuted array of values) implicitly make use of the cyclic decomposition of a permutation. The simplest of these inversion algorithms rather resembles the algorithm for putting a permutation into cycle form.

```
definef cycinv(f); s=hd[f];
(while s ne nl) elt from s; next=f(elt);
(while next∈s) s=s less next; <elt,next,f(next)>
        = <next,f(next),elt>;;
f(next) = elt; /*closing the loop */ end while s; return;
        end cycinv;
```

An even shorter algorithm, due to Boothroyd, may be written as follows.

```
define boothinv(f); s = hd[f]; heads=s;
(∀p∈s) q=p; (while n q≠heads) q=f(q);;
r=f(q); <f(r),f(q)>=<p,f(r)>; heads=heads less r; end ∀p; return;
end boothinv;
```

The analysis of this very short algorithm is surprisingly complex, and may be given as follows. Let f be a cyclic permutation, which we may think of as shifting a group of elements arranged in a circle, each being shifted to the next position around the circle. Flag each element of s as a "head". Then, *process* the elements of s, in any random order, as follows. Take any as yet unprocessed element p, and find the first element q in the sequence $p, f(p), \ldots, f^k(p)$ which is still flagged as a head; drop the flag of $r = f(q)$, and put $f(r) = p$, $f(q)=f(r)$.

To follow the action of the algorithm, we take the set s to be divided into a set of *runs*, where each run consists of a sequence of elements (in their original circular order) beginning

with an element flagged as a head, up to but not including the next element flagged as a head. Then note that by induction we have the following:

i. By definition, the first element of a run is flagged, all others are unflagged.

ii. The last element of a run is unprocessed; the others have all been processed.

iii. For all p of a run but its first element, f(p) is the previous element in circular order; if p is the first element of a run, f(p) is the first element of the next succeeding run (in circular order).

All these remarks hold initially, all runs initially being of unit length. Since by ii) every unprocessed element p is the last element of a run, our procedure will always find the head q of the same run, and by iii) the head r = f(q) of the next run in circular order. By putting f(r) = p, f(q) = f(r), and dropping the flag of r we join two runs into a single run, preserving the properties i, ii, and iii.

This proves that Boothroyd's process works for every cyclic permutation; since every permutation can be decomposed into cyclic permutations, it must work for every permutation.

Next we consider algorithms for the multiplication of permutations given in cyclic form. Suppose that we have a sequence of cyclic permutations

(1) $\quad (\ldots abcd\ldots) \subseteq (\ldots efgh\ldots) \subseteq \ldots \subseteq (\ldots ijk\ell\ldots)$

to be multiplied together. (We continue to assume a 'left-to-right' convention; that is, the first of these maps to be applied to a set s is the *leftmost*.) Then the image of an element a under the product map is obtained by finding the leftmost occurrence of a, and the element b which follows it; then the next occurrence of b further to the right, and the element c which follow it, etc., until the right-hand end of the sequence (1) of cycles is reached. Each occurrence of a symbol x in the product (1) will be used in such an image-finding

subprocess only once; thus if we mark the occurrences that have been used, and continue the image-finding subprocess as long as any positions in the sequence l are still unmarked, we can multiply permutations in cycle form without having to know the set of elements on which these permutations act. The algorithm is most conveniently expressed if, at the end of each cycle, we repeat the first element of the cycle, and mark the corresponding position as used.

The following SETL algorithm, which assumes that a sequence *seqperms* of permutations in cycle form is given, uses the procedure just explained.

```
definef multall(seqperms);
/*first make single sequence with repetitions*/
seq=nl; marked=nl; (1 ≤ ∀n ≤ #seqperms, cyc=seqperms(n))
(1 ≤ ∀m ≤ #cyc) seq(#seq+1)=cyc(m);; seq(#seq+1)=cyc(1);
#seq in marked; end ∀n; result=nl;
(while 1 ≤ ∃[m] < #seq|n mcmarked) /* start new output cycle*/
cyc={<1,seq(m)>}; elt=seq(m+1); loc=m+1; m in marked; [loop:]
(while loc < ∃[n] < #seq|seq(n) eq elt and n nemarked)n inmarked;
elt=seq(n+1); loc=n+1; end while loc;
if elt ne cyc(1) then cyc(#cyc+1)=elt; loc=1; go to loop;
else cyc in result;; end while 1; return result; end multall;
```

We now give another algorithm, which 'in one pass' multiplies permutations given in cycle form. The algorithm exploits the following facts:

i. If * is an object not appearing in a cycle, then the cyclic permutation (ab...cdef) is the following product of maps

$$(f\to *) \subset (e\to f) \subset (d\to e) \ldots \subset (a\to b) \subset (*\to a)$$

ii. If *map* is any mapping, then the composition

$$map' = (x\to y) \subset map$$

is defined by map'(x) = map(y); map'(z)=map(z) if z is different from x.

The SAIL code for this algorithm, which as might be imagined works from right to left, is as follows:

```
definef multall2(seqperms);
map=nl; (#seqperms ≥ ∀n ≥ 1, cyc∈seqperms(n)|#cyc gt 1)
mapstar=if(x is map(cyc(1))) ne Ω then x else cyc(1);
(1 ≤ ∀i < #cyc) map(cyc(i))=if(x is map(cyc(i+1))) ne Ω
        then x else c(i+1);;
map(cyc(#cyc)) = mapstar; end ∀n;
return cycform(map); end multall2;
```

9. Some additional compiler-related algorithms.

In the present section, we give a few additional algorithms which are useful in connection with compilers. The first arises in connection with the EQUIVALENCE declaration which is part of the FORTRAN language. In this language, one is allowed to specify an overlay pattern for arrays by stating that location k in an array A is to be the same as location k' in another array A'; then the j-th address in A' is the address j+k-k' in A. The algorithm, due to Fisher and Galler, that we shall shortly give takes a set of declarations of this sort and divides the set of all arrays mentioned in them into disjoint families, within each one of which every array is related by a given offset to a specified *master array*. In the process, the whole set of declarations is checked for consistency. We write the algorithm as a function, whose input is a set of *declaration triples* <*arraya*, *arrayb*, *offset*>, *offset* being an integer quantity. A flag is set if the whole set of triples reveals an inconsistency. The map *master* gives, for each array A, a pair <*arrayb*, *offset*> consisting of an array to which A is related by a definite offset, and this offset.

```
definef equivproc(dectrips,err);
err=f; master=nl; arrays=nl;
(∀trip∈dectrips) <arra,arrb,offs>=trip;
arra in arrays; arrb in arrays; /*chain to ultimate masters*/
(while master(arra) ne Ω)  /* correcting offsets*/
<arra,offs>=<hd master(arra),offs+tl master(arra)>;;
(while master(arrb) ne Ω)
<arrb,offs>=<hd master(arrb), offs-tl master(arrb)>;;
/* offs is now the offset relating two formerly independent
   master arrays. check for contradiction */
if arra eq arrb and offs ne 0 then err=t; return Ω;;
/* otherwise make second array master of first */
```

```
master(arra) =<arrb,offs>; end ∀trip;
/* now summarize all the relationships that have been collected*/
(∀arra∈arrays)<arr,offs> = <arra,0>;
(while master(arr) ne Ω)
<arr,offs>=<hd master(arr), offs+tl master(arr)>;;
master(arra) = <arr,offs>; end ∀arra;
return master; end equivproc;
```

Next we shall describe the structure of a highly simplified macro-processor of a general type suitable for introducing a macro-feature into a source language. We assume the following facilities.

i. A reserved keyword '*macdef*' is provided. A string of tokens of the form

macdef name arg1 ... argn ,

followed by an end-of-statement mark, introduces a macro definition, the token *name* being designated as the name of the new macro, while the tokens *arg1,...,argn* represent its arguments. The body of the macro then follows, and extends all the way to an occurrence of the token sequence

(1) endmac name

whild should be followed by an end-of-statement mark. The macro body may contain macro-calls and additional macro-definitions.

ii. Once a name has been declared as a macro-name, any subsequent appearance of a string of tokens

name symb1 ... symbn ,

followed by an end-of-statement mark, will cause the body of the macro with the given name to be inserted into a transformed source string, with the original arguments of the macro replaced by the tokens *symb1,...,symbn* respectively. Any argument for which a corresponding token *symbj* is not provided will appear in original form; any surplus argument symbols *symbj* will be ignored.

The algorithm given below maintains an internal 'library' of previously declared macros, in the form of a function *macro* which maps each known macro name onto a unique pair <*argfn*,*mbody*>; here *mbody* is the declared text of the macro, and *argfn* maps each declared prototype argument of the macro onto its serial number.

Our macro-processor is assumed to function as an interface between a parser and the basic lexical *nextoken* routine; the parser calls upon the macro processor when it requires an additional token; the macro-processor either supplies a token produced as the result of a macro-expansion, or, if none is available, calls upon *nextoken* for a new lexical input. Two stacks are used by the macro-processor. One, called *tokstak*, is used for the accumulation of tokens produced when macros are expanded; the other, called *argstack*, is used to save 'outer' macro-arguments during the expansion of inner macro-invocations. A macro is expanded by placing its actual arguments on *argstack* and then transferring a copy of its body to *tokstak*, substituting the actual arguments for the prototype macro arguments during this transfer. A macro-definition is processed by accepting a macro name *mname*, and then accepting all the tokens following the name, up to the next end-record token, as the set of prototype arguments of the macro. All the tokens which then follow, up to an occurrence of the token sequence (1), are then accepted as the macro body. The macro definition is checked to verify that prototype arguments are not repeated; if this test is passed, and the main definition is properly terminated, the argument and macro-body information collected are used either to establish a new macro or to replace an old one.

The SETL code for the macro-processor is as follows.

```
definef mactoken;
initially macro = nl; nargs=0; endrec=<'er',nulo>;
  endfile = <'ef',er+er>; end initially;
[newt:] tok = getoken;  /*tok has form <type,token,data(if any)>*/
  if(<*z2>tok) eq 'macdef' then go to mdef;
    else if (macis is macro(tok)) ne Ω then go to macinv;
      else return tok; end if; end if;
[macinv:] /* macro invocation. stack current number of arguments;
  read new set of arguments */
  nargs stack argstak; tok=getoken; nargs=0;
  (while n tok∈{endrec,endfile}) nargs = nargs + 1;
  tok stack argstack; tok=getoken; end while;
  <argfn,mbody>= macis; /*expand macro*/
  (#mbody ≥ ∀n ≥ 1)
  if(argno is argfn(mtok is mbody(n)) eq Ω
    then mtok stack tokstak;
    else if argno gt nargs /*required argument not supplied*/
      then mtok  stack tokstak;
    else argstack(#argstak-nargs+argno) stack tokstak; end if;
      end if; end ∀n; (1<∀n≤nargs) argstack(#argstack)=Ω;;
                nargs = topoff argstack; go to newt;
[mdef:] /* a new macro-definition to be processed */
  if (mname is getoken)∈{endrec,endfile} then go to macerr;;
  /* next set up argument mapping */
  afn = nl; mok = t; mbody = nl;
  (while n (marg is getoken) ∈ {endrec,endfile})
  if afn(marg) ne Ω then print 'illegal duplication of
    macro-argument'; mok = f;
   else afn(marg) = #afn+1;; end while;
/* now go on to accumulate macro-body */
  (while n (tok is getoken) eq endfile) ne better
    if (<* z 2>tok) eq 'endmac' then tok2 = getoken;
      if tok2 eq mname /*macro is ended; skip
                following end-record mark*/
        then skip = getoken; /*record macro if no errors*/
```

```
        if mok then macro(mname)=<afn,mbody>; go to newt;;
    else /*not end of this macro-ignore*/
        tok stack mbody; tok2 stack mbody; end if tok2;
    else /*not endmacro-ignore*/tok stack mbody; end if;
                                        end while;
macerr: print 'fatal error - improperly terminated
                    macro definition'; exit;
definef getoken; /* takes token from token stack or from input*/
return if tokstak ne nl then topoff tokstak else nextoken;
                                        end getoken;
definef topoff stack; elt = stack (#stack); stack(#stack)=Ω;
    return elt; end topoff;
define elt stack stk; stk(#stk)=elt; return; end stack;
/* end of macro-processor */
end mactoken;
```

10. Table and data compaction algorithms.

In any computer system, storage space and especially high-quality storage, is available only in limited amounts. For this reason the problem of table and data compaction, i.e., the problem of encoding a given mass of data into the most compact form possible, is an important one. This problem arises also in connection with the transmission of data; in order to maximize the rate at which useful information is transmitted, we wish to make use of an encoding which reduces to a minimum the number of bits which must be transmitted. This compression problem has been much studied, and has generated a vast literature; we shall in the present short section touch on only a very few of the interesting ideas which have been developed. Generally speaking, data is compressed by discovering and exploiting regularities in it; the most drastic compressions will come when a large table of data can be replaced by a short routine capable of calculating all its entries. Even where this is not possible, any statistical regularity which the data exhibits may be used to secure compression. The first set of algorithms we shall exhibit, those for *Huffman coding*, make use of this idea.

The setting assumed is as follows. Text, consisting of a stream of characters, is to be encoded. The various possible characters occur with differing relative frequencies, the expected frequency of character c being given by *freq*(c).

It is then advantageous to assign a binary code to each character in such a way that the most probable characters receive short codes, while the least probable characters receive long codes. In this way we may on the average expect a text to assume a compact representation. Huffman's specific technique is as follows: take the two characters c,d of smallest frequency, and hang them as left and right branches from a newly created node n, whose heuristic meaning is 'either c or d'. Remove c and d from the set of characters,

and insert n, taking its frequency to be the sum of that of c and d. Repeat this operation until only a single character remains, in the process growing a tree, the so-called *Huffman tree* of the set of characters. The code for a character is then its address in this tree, where 'go down to the left' is represented by a binary 0, and 'go down to the right' is represented by a binary 1. The decoding process to be applied to a stream of characters represented in Huffman coded form by a binary sequence is then clear: Start at the top of the tree, and proceed downward to a twig, in the manner directed by the binary sequence. When a twig is reached, take the character found there as input, jump back to the top of the tree, and repeat.

In SETL, the Huffman-tree and encoding-table build routine appears as follows:

```
definef huftables(chars,freq); work=chars; wfreq=freq; ℓ=nℓ;r=nℓ;
(while #work  gt 1) c1=getmin work; c2=getmin work; n = newat;
ℓ(n)=c1; r(n)=c2; wfreq(n)=wfreq(c1)+wfreq(c2); n in work;
     end while;
code=nℓ; seq=nul; walk(top is work); return <code,ℓ,r,top>;
definef getmin set; huftables external freq;
     <keep,least>=<x is set,freq(x)>;
     (∀x∈set) if freq(x) lt least then <keep,least>=<x,freq(x)>;
end ∀x; return keep; end getmin;
define walk(top); /*recursive tree-walker which builds up
        address of each twig*/
huftables external code, seq;
if ℓ(top) ne Ω then seq = seq+0b; walk(ℓ(top));
        seq = seq+1b; walk(r(top));
else /* at twig */ code(top) = seq; end if;
seq = len seq-1 first seq; return; end walk;
        end huftables;
```

Next, the decoding process. Given a sequence *bitseq* of bits, we produce a sequence *cseq* of characters by the following decoding algorithm.

```
definef cseq(huftables,bitseq); <-,ℓ,r,top>= huftables;
output=nl; node=top; (1 ≤ ∀n ≤ #bitseq)
if ℓ(node) eq Ω /*so that we are at twig*/
  then output(#output+1) = node;
    node=top; else node=if bitseq(n) eq 0
        then ℓ(node) else r(node); end if;
  end ∀n; return output; end cseq;
```

The basic Huffman-coding technique that we have outlined can be refined in a number of ways. Instead of encoding single characters, we can encode pairs or triples of characters using the Huffman technique. Instead of using a fixed set of frequencies, we can collect separate tables showing the relative frequency which each character will have in known preceding contexts, e.g. the relative frequencey which c will have in those cases when it follows immediately after a given character d. This information can then be used to build up a set of Huffman tables, one for each context that we wish to distinguish. During the decoding process, we will of course know the preceding context of each symbol s we are attempting to decode, having decoded the symbols determining this context before we begin trying to decode s. We leave it to the reader to work up generalized Huffman coding and decoding algorithms incorporating these more sophisticated possibilities.

Next we consider the problem of representing a *binary relationship* over a given set s. Such a relationship is a 0,1-valued function r(x,y), defined for all x∈s, y∈s. A relationship of this kind can of course be represented (at machine level) by a table of bits, n×n in size, where n is the number of elements of s. Various other possibilities,

which may in certain cases be more efficient, exist. For example, we may associate with each x the set of all y for which $r(x,y) = 1$. Here we shall consider another way of representing $r(x,y)$, which may be used with advantage in certain cases. Our idea is as follows. First associate with each x the set

$$U_x = \{y \in s \mid r(x,y) \text{ eq } 1\}.$$

If the family of sets U_x can be arranged in a single, steadily increasing family $U_{x_1} \subseteq U_{x_2} \subseteq U_{x_3} \subseteq \ldots \subseteq U_{x_n}$ of sets, then we can assign, to each x in s, the index of the set U_x in this sequence, thereby defining a function f(x). If, for each y, we let g(y) be the index of the smallest set U_x to which y belongs, then plainly r(x,y) eq 1 is equivalent to $g(y) \leq f(x)$. Hence r(x,y) can be represented by a table of 2n values, rather than by a table of n^2 bits.

In general, we cannot expect the sets U_x to form a single, steadily increasing chain. Nevertheless, the family of all sets U_x will be partially ordered (cf. pp. 132-133) and we can use Dilworth's theorem (see the algorithm of p. 134) to break the collection of all the sets U_x into a minimum number of separate steadily increasing chains of sets. We may then let f(x) be the index of U_x in the chain c(x) to which U_x belongs, and, for each y∈s and each chain c, let g(y,c) be the index of the smallest set, in the chain c, which contains y. It then follows that r(x,y) eq 1 is equivalent to $g(y,c(x)) \leq f(x)$. If k chains are needed to accommodate all the sets U_x, this method allows us to represent the relation r(x,y) by a table of (k+2)n, rather than by a table of n^2, values.

This compaction procedure, quite useful in certain cases, may be written as follows in SETL.

```
definef compactreln(set,relation);
relpairs= {<x,y>, xeset, yeset|∀zeset|relation(y,z)
                    imp relation(x,z)};
definef reln(x,y); compactreln external relpairs;
  return <x,y>erelpairs and x ne y; end reln;
/* note that an external function named reln is used
  by the algorithm on page 134, which we now invoke*/
chains=dilworth(set); f=nl; g=nl; c=nl;
(∀chechains, 1 ≤ j ≤ #ch) f(ch(j)) = j; c(ch(j))=ch;;
(∀yeset, chechains) g(y,ch)=if  1 ≤ ∃[j] ≤ #ch|relation(ch(y),y)
          then j else #ch+1;;
return <f,g,c>; end compactreln;
```

12. Symbolic manipulation: some theorem-prover algorithms.

If algorithms are to discover and perform many of the powerful simplifying transformations utilized by human programmers, they will have to be able to find those proofs of correctness with which a programmer is able to justify such transformations. This is but one of the many considerations which lend interest to algorithms which find proofs of mathematical theorems. Such algorithms, some of which we shall present in the present section, are of two rather different types. One type is designed specifically for a given field of mathematics (such as Euclidean plane geometry), and incorporates structures and procedures particular to this field. The other type of theorem-prover, which is the type with which we shall concern ourselves, works with general methods taken from mathematical logic, and can in principle prove theorems in any field if given axioms describing that field. (However, if the algorithm is lacking in efficiency, entirely impractical amounts of time and memory may be required.)

Any algorithmic scheme for proving theorems must of course rest upon a formalization of the part of mathematics which is to be covered, i.e., upon a method for the representation of a mathematical subject domain by a set of formulae subject to algorithmic manipulation. To this end, the algorithms which we shall consider make use of the so-called *first order functional calculus* of mathematical logic. In this calculus, formulae are built up from the following elements:

i. The basic *boolean connectives*: *and*, *or*, *not*, *implies*, *is equivalent to*; or as logicians normally write them

$$\&\ ,\ \vee\ ,\ \sim\ ,\ \supset\ ,\ \equiv$$

ii. *Variables*, representing objects varying over a given mathematical domain. Normally written x, y, etc.

iii. *Constants*, representing particular distinguished objects of a mathematical domain. Normally written c, d, etc.

iv. *Function symbols*, representing particular basic or derived mappings of objects to objects in a mathematical domain. Normally written f, g, etc. Each function symbol is understood to have some particular characteristic number of arguments. By supplying arguments to a function symbol, we can form symbolic structures representing constant or variable objects, as in the following examples:

$f(x)$; $f(c)$; $g(x,y)$; $g(c,f(x))$; $g(c,f(f(g(f(y),g(y,d)))))$.

v. *Predicate* symbols, representing particular functions whose values are all either $\underline{t}$ or $\underline{f}$. Normally written P, Q, etc. Supplying a predicate with its fixed number of arguments, we form *units* representing constant or variable boolean values, e.g.

$P(f(x))$; $Q(c,c)$; $Q(g(x,y),f(x))$; $Q(g(c,y),g(y,f(c)))$.

vi. The boolean quantifiers $(\forall x)$, $(\exists x)$, where x denotes any variable. By combining units with connectives and quantifiers in syntactically valid ways, we form all the allowed formulae of the first order functional calculus. Examples of such formule are

$(\forall x)f(x) \supset f(c)$; $g(c,c) \supset (\exists x)(\exists y)(g(x,y) \vee \sim f(x))$.

The formulae of the first order functional calculus are related to each other by a formal notion of proof. Any set Σ of formulae of the functional calculus may be designated as a set of hypotheses. A proof from Σ is a sequence ϕ_i of formulae, each of which is either

i. An axiom;

ii. A formula of Σ;

iii. A consequence of some prior formulae by the rules of elementary boolean logic. More precisely, suppose that ϕ_1, ϕ_2 and ψ are formulae containing proposition-symbols and boolean connectives only, and that ψ follows from

ϕ_1 and ϕ_2 by the rules of elementary boolean logic. Then, if substituting a term for each of the proposition symbols in ϕ_1, ϕ_2 and ψ we produce three formulae ϕ_1', ϕ_2' and ψ' of the first order functional calculus, we say that ψ' is a conseauence of ϕ_1' and ϕ_2' by the rules of elementary boolean logic.

iv. A formula derived froma prior formula ϕ_i by changing the names of the variables in ϕ.

v. Some formula of the form

(1) $$(\forall x)\alpha(x) \equiv \sim(\exists x)\sim\alpha(x) ,$$

where $\alpha(x)$ is a formula not containing either $(\forall x)$ or $(\exists x)$.

vi. A formula either of the form

(2) $$(\forall x)\alpha(x) \supset \alpha(y) ,$$

where $\alpha(x)$ is a formula not containing y, $(\forall x)$, or $(\exists x)$; or a formula of the type

(2') $$(\forall x)\alpha(x) \supset \alpha(f(y_1,\ldots,y_n)) ,$$

where $\alpha(x)$ is a formula not containing $y_1,\ldots,y_n$, $(\forall x)$, or $(\exists x)$.

vii. A formula derived from a prior formula ϕ_i of the form

(3) $$\alpha \supset \beta(x)$$

where α does not contain x, by transforming (3) into

(4) $$\alpha \supset (\forall x)\beta(x) .$$

We may use the formal notion of proof just described to formalize quite arbitrary parts of mathematics, as follows. We introduce function symbols and predicate symbols to represent all the basic constructions and relations characterizing a given mathematical domain. Then, using these symbols, we write out all the axioms of the subject. A proposition written in these same symbols will be a theorem if and only if it can be derived from these axioms in the formal sense just defined.

We now note that the familiar principle of *proof by contradiction* can readily be established within our formal logical calculus: a theorem ϕ can be proved from hypothesis Σ if and only if a contradiction, i.e., some pair of formulae of the forms ψ and $\sim\psi$, can be derived from the union set Σ with $(\sim\phi)$. Thus any algorithm capable of deciding when a given set of logical formulae leads to a contradiction can be used as a theorem-prover. This is the basic approach which all the algorithms to be described in the present section will use. They will in fact generally embody a rather more specific strategy, which can be put as follows. First of all, we may use the laws of Boolean logic to eliminate all connectives but &, V, and $\sim$ from a set of logical formulae. In particular, $A \supset B$ can be changed to $\sim A \vee B$, and $A \equiv B$ can be changed to $(A\&B) \vee (\sim A \& \sim B)$. Then using the following rules, we may move all negation signs 'inward' until every remaining negation sign immediately precedes a predicate symbol:

(5)
$\sim(\forall x)A$ is equivalent to $(\exists x)\sim A$;
$\sim(\exists x)A$ " $(\forall x)\sim A$;
$\sim(A\&B)$ " $\sim A \vee \sim B$;
$\sim(A \vee B)$ " $\sim A \& \sim B$;
$\sim\sim A$ " A .

Next, make the following observation. Heuristically speaking, the truth of a compound proposition like

(6) $(\forall x)(\forall y)(\exists z)\ P(x,y,z)$,

which we may think of as constituting part of some longer formula G which we wish to analyze, is equivalent to the truth of

(7) $(\exists f)(\forall x)(\forall y)\ P(x,y,f(x,y))$,

where f denotes a "choice-function" which for each x and y chooses a particular z satisfying (6). Of course, in writing (7), we have deviated from the strict formalism of the first order functional calculus, which does not allow quantification

of function symbols. Nevertheless, if for the moment we allow ourselves to pursue this heuristic train of thought, we may reason as follows. Since

(8) $((\exists f)\ A(f)) \vee B$ is equivalent to $(\exists f)\ (A(f) \vee B)$

and $((\exists f)\ A(f)) \& B$ " $(\exists f)\ (A(f) \& B)$

if B does not contain any occurrence of f, all the quantifiers $(\exists f)$ generated in passing from (6) to (7) can be moved to the front of the transformed formula G' containing (7). Then, noting that rules like (8) also apply to universal quantifiers, we see that after moving all existential quantifiers $(\exists f)$ to the front of G', it will also be possible to move all universal quantifiers to the front of G'. Thus, from an original proposition G, we can produce one of the form

(9) $$(\exists f_1) \ldots (\exists f_n)\ (\forall x_1) \ldots (\forall x_m)\ A(x_1 \ldots x_m)\ .$$

Now, since the prefixed existential quantifiers in (9) merely assert the existence of certain functions about which nothing else is known, we can drop these quantifiers, merely retaining the function names $f_1, \ldots, f_n$ in our formulae as additional function names with our transformation process will have generated. This step however brings us back to a formula G" of the first order functional calculus, in which new function symbols will have appeared, and in which all remaining quantifiers are universal and appear at the front of the formula. We have then reason to believe that if a contradiction can be derived from our original formula G, a contradiction can also be derived from its transformed version G", and conversely. This conjecture can in fact be established rigorously, as a formal assertion concerning the first order functional calculus. The formula G" is called the *Herbrand normal form* of G. Thus a theorem-prover which works by the method of contradiction can as well begin from the Herbrand normal form of a set of propositions as from these propositions in their original form.

Let us review the process by which a set $G_1,\ldots,G_m$ of formulae is converted to Herbrand normal form, carrying this a few steps further than we have done above. For each G_j, we

a. use the laws of boolean logic to eliminate all connectives other than &, V, and ~.

b. use the rules (5) to move all negation signs to positions immediately preceding predicate symbols.

c. If z is an existentially quantified variable occurring within the scope of certain universally quantified variables $x_1,\ldots,x_n$, we replace z by an element of the form $f(x_1,\ldots,x_n)$, where f is a newly created function symbol.

d. Then we move all universal quantifiers to the front of the resulting formulae, renaming quantified variables when necessary, and using the rules

(10) $(\forall x)\ A(x)\ \&\ B$ is equivalent to $(\forall x)\ (A(x)\&B)$
$(\forall x)\ A(x)\ \vee\ B$ " $(\forall x)\ (A(x)\vee B)$

which apply when B does not contain x.

e. Next, we use the boolean distributive law to write all the resulting formulae in the expanded form

(11) $(\forall x_1)\ldots(\forall x_n)(A_1(x_1\ldots x_n)\ \&\ A_2(x_1\ldots x_n)\ \&\ \ldots\ A_m(x_1\ldots x_n))$,

where the formulae A_i contain no quantifiers, and contain only the boolean connectives V and ~, but not &.

f. We may then separate each formula (11) into several formulae

(12) $(\forall x_1)\ldots(\forall x_n)\ A_j(x_1,\ldots,x_n)$.

g. Finally, we drop the universal quantifiers in the formulae (12), thereby arriving at a set of formulae, each of the form

(13) $t_1(x_1,\ldots,x_n)\ \vee\ t_2(x_1\ldots x_n)\ \vee\ \ldots\ \vee\ t_p(x_1,\ldots,x_n)$,

where every unit t_i is built up from the free variables indicated, and from constants, function symbols, and a predicate symbol, and is either prefixed by a single negation sign or not.

A formula of the special form (13) is called a Herbrand *clause*; the full set H of clauses (13) produced from an original set of formulae $G_1,\ldots,G_m$ by application of steps a) through g) is called the *Herbrand normal form* of the set $G_1,\ldots,G_m$. As we have indicated, the set $G_1,\ldots,G_m$ is *inconsistent*, i.e., can be used to derive a contradiction, if and only if the set H of Herbrand clauses (13) constituting its Herbrand normal form can be used to derive a contradiction.

The so-called *Herbrand theorem*, sometimes dignified by being called *Herbrand's fundamental theorem*, now tells us that a contradiction will only follow from the clauses H if such a contradiction can be derived in a certain quite particular way. Take all the clauses of H, and in each of them substitute, for each variable element, elements built from the available constants and function symbols, making these substitutions in every way possible, and thereby obtaining a countably infinite set H' of substituted formulae. (Note that by the rules of the first order functional calculus all of the formulae of H' are consequences of the formulae of H.) Then Herbrand's theorem states that a contradiction can be derived from the formulae of H (by the rules of first order functional calculus) if and only if a contradiction can be derived from the formulae of H' *by the rules of elementary boolean logic*.

This last statement gives the key to the structure of most of the theorem-proving algorithms which we shall consider; they work by generating formulae belonging to the set H', and searching for a boolean contradiction (which may of course be detected algorithmically without any difficulty). Since H' is infinite, it is of course essential that this search be conducted in an intelligent, i.e., efficient manner, though in purely abstract principle it would be sufficient to generate all of H' progressively in some systematic way, say by substituting elements for the variables of H' simply in increasing order of the length of these elements. We shall discuss the vital question of search efficiency somewhat below. However, before doing so, an example may be helpful. Consider the following simple theorem, often used

as an example in connection with theorem-proving algorithms: an associative semi-group with identity and left inverses is a group, i.e., the left inverse is also a right inverse.
The Herbrand clauses applicable in this case are:

i.	$E(x,x)$	axioms for equality
ii.	$\sim E(x,y) \vee E(y,x)$	
iii.	$\sim E(x,y) \vee \sim E(z,y) \vee E(x,z)$	
iv.	$\sim E(y,x) \vee \sim E(y,z) \vee E(x,z)$	
v.	$\sim E(x,y) \vee E(m(x,z),m(y,z))$	
vi.	$\sim E(x,y) \vee E(m(z,x),m(z,y))$	
vii.	$\sim E(x,y) \vee E(i(x),i(y))$	
viii.	$E(m(m(x,y),z),m(x,m(y,z)))$	semigroup axioms
ix.	$E(x,m(x,e))$	
x.	$E(x,m(e,x))$	
xi.	$E(m(i(x),x),e)$	assumption of left inverse
xii.	$\sim E(m(c,i(c)),e)$	denial of theorem

The clarity of what we have now to do will be enhanced if we allow ourselves the use of infix notation, writing $E(x,y)$ as $x=y$ and $m(x,y)$ as $x \cdot y$. Using this notation, we proceed as follows:

xiii. Generate the formula $i(i(c)) \cdot i(c) = e$, which we call P, from xi by substitution.

xiv. Similarly generate $\sim[i(i(c)) \cdot i(c) = e] \vee [(i(i(c)) \cdot i(c)) \cdot c = e \cdot c]$, which we call $\sim PVQ$, from v.

xv. From the last two propositions, generate Q, i.e., $(i(i(c)) \cdot i(c)) \cdot c = e \cdot c$ by Boolean logic.

xvi. Generate $(i(i(c)) \cdot i(c)) \cdot c = i(i(c)) \cdot (i(c) \cdot c)$ from viii, and call it R. Then

xvii. Generate $\sim Q \vee \sim R \vee S$, i.e.

$$\sim[(i(i(c))\cdot i(c))\cdot c = e\cdot c] \vee \sim[(i(i(c))\cdot i(c))\cdot c = i(i(c))\cdot(i(c)\cdot c)] \vee [i(i(c))\cdot(i(c)\cdot c) = e\cdot c]$$

from iv.

xviii. From this last proposition, and from R and Q, generate S, i.e.

$$i(i(c))\cdot i(c)\cdot c) = e\cdot c$$

by Boolean logic.

xix. Generate

$$i(c)\cdot c = e \ ,$$

which we call T, from xi.

xx. Generate $\sim TVU$, i.e.,

$$\sim[i(c)\cdot c=e] \vee [i(i(c))\cdot(i(c)\cdot c) = i(i(c))\cdot e]$$

from v, and use this and T to generate U, i.e.

$$i(i(c))\cdot(i(c)\cdot c) = i(i(c))\cdot e$$

by Boolean logic.

xxi. From this last proposition, from xviii, and from

$$\sim[i(i(c))\cdot(i(c)\cdot c)=e\cdot c] \vee \sim[i(i(c))\cdot(i(c)\cdot e)=i(i(c))\cdot e] \vee [e\cdot c=i(i(c))\cdot e]$$

which is a substituted case of iv, derive the proposition V, i.e.

$$e\cdot c = i(i(c))\cdot e$$

by Boolean logic.

xxii. Generate

$$i(i(c)) = i(i(c))\cdot e$$

from ix. From this, from the preceding proposition, and from

$$\sim[i(i(c))=i(i(c))\cdot e] \vee \sim[e\cdot c=i(i(c))\cdot e] \vee [i(i(c))=e\cdot c] \ ,$$

which is obtained from iii by substitution, obtain

$$i(i(c)) = e\cdot c$$

by Boolean logic.

xxiii. Generate

$$c = e \cdot c$$

from x. From this, from the preceding proposition, and from

$$\sim[c=e\cdot c] \vee \sim[i(i(c))=e\cdot c] \vee [c=i(i(c))]$$

obtain

$$c = i(i(c))$$

by Boolean logic.

xxiv. From this last proposition, and from

$$\sim[c=i(i(c))] \vee [c\cdot i(c)=i(i(c))\cdot i(c)] ,$$

which is obtained from v by substitution, get

$$c\cdot i(c) = i(i(c))\cdot i(c)$$

by Boolean logic. From xiii, and using the substituted version

$$\sim[i(i(c))\cdot i(c)=e] \vee [e=i(i(c))\cdot i(c)]$$

of ii, obtain

$$e = i(i(c))\cdot i(c)$$

by Boolean logic.

From these last two propositions, and using the substituted version

$$\sim[c\cdot i(c)=i(i(c))\cdot i(c)] \vee \sim[e=i(i(c))\cdot i(c)]$$
$$\vee \quad c\cdot i(c) = e ,$$

of iii obtain

$$c\cdot i(c) = e$$

by Boolean logic.

xxv. The proposition obtained in xxiv is precisely the negative of xi, completing our proof.

This somewhat lengthy example will illustrate the manner in which boolean contradictions can be derived by substitution. As has been remarked, the substitution process is, setting aside all considerations of efficiency, straightforward. Starting

from our original clauses, we generate a larger and larger mass of clauses by substitution and boolean deduction. If our procedure is exhaustive and our original clauses actually inconsistent, Herbrand's theorem tells us that a contradiction must eventually emerge. Of course, if we are to have any practical chance of finding this contradiction, we must use a method more selective than wholesale formula generation. Any theorem-prover algorithm worth considering will in fact incorporate some heuristic for formula selection, which aims to reach a contradiction before the mass of propositions generated becomes overwhelmingly large. Heuristics of at least two types are conceivable. We might imagine heuristics which examine a set of propositions for subsets exhibiting particular patterns, and, on finding these, route the course of formula generation in particular directions. This corresponds to part of the normal procedure of the mathematician, who, on examining the formulae given above, would probably note that 'transitivity' and 'associativity' both occur, and, having made this observation, would employ a special notation permitting deduction to proceed in particular directions more rapidly than does the completely general stepwise substitution process that we have illustrated.

Such 'feature noticing' heuristics have in fact not been used much in general theorem-proving algorithms of the kind we shall consider; the development of methods of the kind suggested is probably a useful direction of future work. Algorithmic theorem provers have instead used heuristics of a more general and formal character. Such heuristics can be developed through formal analysis of the manner in which contradictions might be derived. The following reasoning belongs to this line of thought, and will serve to illustrate what is meant. In deriving a contradiction, we may restrict ourselves to any method of Boolean deduction which is capable of making manifest the contradictory character of any set of inconsistent Boolean clauses. It is not hard to see that to this end we may rely exclusively on use of the

following Boolean rule: from two clauses a ∨ B and ¬a ∨ C, where B and C are themselves subclauses, derive B ∨ C. This Boolean rule is sometimes called *resolution*. Now, in the situation with which we mean to deal, the basic terms which can be involved in such resolutions will be obtained by substitution of particular elements for the variables occurring in an originally given set of clauses. Given that resolution is the sole Boolean technique that we shall employ, it is clearly not worth making any substitutions which do not lead to the production of a pair of formulae of the form a ∨ B and ¬a ∨ C. On the other hand, given a pair of formulae $a_1 \vee B'$, $\neg a_2 \vee C'$, we can tell algorithmically whether there exists any substitution which, in the necessary sense, will *unite* a_1 and a_2. For this to be possible, the units a_1 and a_2 must begin with the same predicate symbol, and must be structured in corresponding ways: whenever a_1 and a_2 differ, a variable x must be found in one where a variable, constant, or compound structure e occurs in the other. The most general substitution which unites a_1 and a_2 is then the substitution which replaces each such x by the corresponding e. For example, the units

$$P(x,f(y)) \quad \text{and} \quad P(g(z),w)$$

can be united (made identical) by putting g(z) for x and f(y) for w; this is the most general substitution which can be used to make these two units identical.

The above analysis makes it clear that the following set of rules will derive a Boolean contradiction from a set of clauses, if such a contradiction can be derived.

1. Search the clauses for pairs $a_1 \vee B'$, $\neg a_2 \vee C'$ such that a_1 and a_2 can be united by making some substitution σ for the variables occurring in these clauses.
2. Make this substitution in both clauses, producing two clauses a ∨ B, ¬a ∨ C to which Boolean resolution can be applied.
3. Apply resolution, thus adding a new clause B ∨ C to the stock of clauses available.
4. Iterate, until a null clause, i.e., a contradiction, emerges.

The procedure we have outlined, which essentially describes the first formal algorithmic method we shall consider, the so-called *binary resolution method*, will find a contradiction if one exists. In this sense, it is *complete*. But by its very nature it bypasses many of the useless false starts which a cruder substitution method might make, only generating substitutions which have some advantage, at least locally. This shows the way in which general formal analysis can suggest improvements in the efficiency of theorem proving algorithms. Of course, the preceding paragraphs have by no means exhausted the measure of formal analysis which may be brought to bear. It is useful, for example, to consider the pattern in which negated and unnegated terms in clauses can enter into the derivation of a contradiction. We will take up this point somewhat below: it will lead us to the so-called *hyper-resolution* variant of the binary resolution method. It may also be observed that, in addition to 'global' heuristics derived by consideration of all the possible ways in which a contradiction might emerge, one may also find 'local' heuristics of a somewhat more *ad-hoc* character to be useful. For example, long clauses will in many cases not be worth considering, since such clauses must either contain many terms, whose eventual elimination by successive resolutions may be doubtful, or highly substituted and hence very specific terms, which it may be difficult to match with terms in other clauses. For the reasons suggested, some heuristic scheme which counts the number of characters which a clause contains and prefer short to long clauses may be advantageous. We may also wish, as we generate clauses c_i, to attach to c some measure of the number of steps required to derive c. We might then set aside all c for which this measure grew too large, thereby giving preference to 'shallow' proofs over 'deep' proofs; most of our algorithms are incapable of finding deep proofs in any reasonable length of time, and hence the hint that a proof is shallow is very useful. In point of fact, we shall generally not incorporate heuristics of *ad-hoc* character into the algorithms to be presented in this section. We prefer to omit them in order that the various strategies derived from formal

analysis which are to be presented can stand forth with maximum clarity. We shall however give one example of a basic theorem-prover algorithm with *ad-hoc* heuristics attached, so as to show how this is to be done.

It should also be noted that the algorithms which follow simply derive a contradiction (if they can find it in times short of astronomical) and report their success. Actually, algorithms intended as more than sketches of method would keep an account of the parentage of each clause, and, on deriving a contradiction, would print out the ancestry of this contradiction, thus exhibiting the proof that they have found, rather than merely asserting success. As this addition to the algorithms which are to follow is easily made, we have felt free to omit it.

After these general introductory remarks, we begin a more detailed algorithmic discussion. We will always start from a set of clauses which are to be shown to be contradictory. We represent each clause c by a pair

<negunits,posunits> ,

where *posunits* is the set of all units which occur unnegated in c, and *negunits* is the set of all units which occur negated in c. Each unit itself is taken to be a pair,

<relnsymbol,argsequence> ,

where *relnsymbol* is some (atomic) relation symbol, and *argsequence* is a sequence, namely the sequence of arguments of this relation. Finally, each argument is an element, i.e., is either

i. A variable, represented by an atom;

ii. A constant, represented by a set whose sole member is an atom;

iii. An *item*, represented by a pair

<functionsymbol,argsequence> ,

where *functionsymbol* is some (atomic) function symbol, and *argsequence* is the sequence of arguments of the function, having again one of the forms i, ii, or iii. All of this is to say that we represent clauses, and their units and elements, by particular recursively defined, tree-like structures.

Before coming to the algorithms of principal concern to us, we build up a group of auxiliary routines for the manipulation of logical formulae represented in the form we have just described. First, some elementary algorithms:

```
definef a lesset b; /* the difference-set operation */
return {x∈a|n x∈b}; end lesset;

definef varsof(obj); /* a recursive tree-walker which collects
all the variables of a term or element */
if atom obj then return obj;;if  n pair obj then return nl;
   else return [u: 1≤n≤#(aseqis tl obj)]varsof(aseq(n));;
end varsof;
```

Next, two subroutines which implement basic substitution operations.

In these basic algorithms (and in certain subsequent algorithms as well) we take a *substitution* to be defined by a mapping

(1) $$map = \{\langle x_1, e_1\rangle, \langle x_2, e_2\rangle, \ldots, \langle x_n, e_n\rangle\},$$

which associates some unique element with each of a finite set of variables x_j. The effect of the substitution (1) on a term is to replace all the variables x_i simultaneously by the element e_i; variables x for which map(x) eq Ω are not replaced. Note then that nl acts as the identity substitution.

```
definef obj subst map; /*the basic recursive substitution
operation, applying either to a term or an element*/
return if atom obj then if(new is map(obj)) ne Ω then new
  else obj
  else if pair obj then <hd obj,{<n,(tl obj)(n) subst map>,
    1 ≤ n ≤ #tl obj}>;
  else obj;  end subst;
```

```
definef mapa compose mapb; /*the 'multiplication' of maps
 corresponding to the preceding substitution operation*/
return {<hd x, tl x subst mapb>, xεmapa} ∪
 {item ε mapb | ∩ hd item ε hd{mapa}}; end compose;
```

Two units (or elements) *ta* and *tb* are said to be *unifiable* if there exists a substitution *map* such that

(2) (*ta* subst *map*) eq (*tb* subst *map*) .

If such a map exists, then there exists a *most general unifier*, i.e., a map m with the property (2) such that any other map with the property (2) can be written in the form m compose n for some n. The map m can be found by the following procedure: *ta* and *tb* must be structurally identical, except that where a variable x occurs in either, some compound element e may occur in the other. If such an x can be found, substitute e for each occurrence of x, thus bringing *ta* and *tb* closer to equality, and continue to search for additional variables x for which substitutions ought to be made. The map we desire is then the composition of all the individual substitutions which this process will uncover. Note however that if x itself occurs as a variable in the element e to be substituted for it, then unification becomes impossible.

The following SETL algorithm makes use of the procedure just described. The algorithm is written as a function which returns the most general unifier of two units (or elements); if unification is impossible, it returns Ω.

```
definef mogenu(ta,tb); if ta eq tb then return nl;;
if atom ta then return
     if tabvarsof(tb) then Ω else {<ta,tb>};;
if atom tb then return
     if tbavarsof(ta) then Ω else {<tb,ta>};;
if ¬ pair ta or ¬ pair tb then return Ω;;
if #argsa>ta ne <*,argsb>tb then return Ω;;
map=nl; /*the identity substitution*/ (1≤∀i≤#argsa)
objb=argsa(i) subst map; objb=argsb(i) subst map;
if (mgu is mogenu(objb,objb)) eq Ω then return Ω;
          else map=map compose mgu;; end ∀i;
return map; end mogenu;
```

Next we give a routine which forms part of the simplification procedure we will use. A clause c is said to *subsume* a clause c' if there exists a substituted instance $\bar{c}$ of c, such that every positive unit of $\bar{c}$ is included among the positive units of c' and every negative unit of $\bar{c}$ is included among the negative units of c'. In this case it is clear that c represents a more general assertion than c', so that c' may be dropped without loss of information from any set of assertions containing c. Let the units of c' be

$$t'_1 \vee t'_2 \vee \ldots \vee t'_n ,$$

and the units of c be

$$t_1 \vee \ldots \vee t_m .$$

We test for subsumption as follows. Taking t_1, we form the collection of all maps which would make t_1 identical with a term of c'. Then, taking each such map m, we attempt to *extend* it to the next unit t_2 of c, by the following rule. Take t_2, and apply the substitution m to it; call the result $\bar{t}_2$. If there is some map m' which makes $\bar{t}_2$ identical with a unit of c', the composition m compose m' represents the desired extension of m; if no such m' exists, we simply drop m. Any maps which can be extended to t_2 we attempt to extend to t_3 etc. If there exists any map which can be extended to all the units of c, then c subsumes c'; otherwise not.

Note that in our extension process, we never wish to make a substitution for any variable in a unit of c which prior substitutions have made identical with a unit of c'. This part may be handled neatly by redesignating all the variables of c' as constants before the extension process begins. Given this, the compositions of maps to be performed during the extension process reduce simply to unions of sets of ordered pairs.

Here is the SETL algorithm.

```
definef c subsumes cpr; cprime=cpr;
varspr=[u: t∈hd cprime u tl cprime] varsof(t);
makeconst= {<x,{x}>, x∈varspr};
cprime = <[hd cprime] subst makeconst,[tl cprime] subst makeconst>;
maps={nl}; /*start with set consisting of identity map only*/
<neg,pos>=c; <negpr,pospr>= cprime;
(∀t∈neg) if (maps is extend(maps,t,negpr) eq nl then return f;;;
(∀t∈pos) if(maps is extend(maps,t,pospr)) eq nl then return f;;;
/*if arrive at this point then*/ return t;
definef extend(maps,unit,unitset); extensions=nl;
(∀m∈maps,t∈unitset) if(newmap is mogenu (unit subst m,t))ne Ω
  then newmap u m in extensions;; end ∀m; return extensions;
end extend;  end subsumes;
```

We will also have use for the 'pluralized' variant of the above procedure in which we ask whether any one of a set of clauses subsumes a given clause. This is expressed in SETL simply as follows.

```
definef clauses subsume c; return ∃clause∈clauses|clause
        subsumes c; end subsume;
```

Now we come to the resolution process itself. Suppose that, given clauses ca and cb we can find a negative term na in ca, and a positive term pb in cb with which na can be unified. Let m be the most general unifier of na and pb. The *resolvent of* ca *with* cb *on* na *and* pb is then obtained by dropping na from ca, pb from cb, joining the remaining terms into a single clause, and applying the substitution m to the result. In SETL, this is:

```
definef resolvent(na,pb,ca,cb); m=mogenu (na,pb);
hcasub=[hd ca] subst m;   tcasub=[tl ca] subst m;
        hcbsub = [hd cb] subst m;
tcbsub = [tl cb] subst m; nasub=na subst m;
return <hcasub less nasub u hcbsub; tcbsub less nasub u tcasub>;
end resolvent;
```

A set s of clauses produced by resolution may contain clauses c subsuming other clauses d in s. It may also contain *tautologous* clauses, i.e. clauses containing a given term both negated and unnegated. Such clauses can clearly never contribute to a contradiction, and may therefore be dropped. The simplification procedure which we now give accepts a set of old and a set of new clauses, removes all subsumed clauses, tautologous clauses, and superfluous units from both sets, and ensures that the variables occurring in any new clauses are disjoint from those appearing in any other clause. This last is a condition assumed implicitly in some of our other algorithms.

```
define simplify(newclauses,oldclauses);
news = newclauses; olds=oldclauses;
(∀clause newclauses) news=news less clause;
if n superfluous(rebuild(clause)) then rebuild(clause) in news;;
   end ∀clause;
(∀clause ε oldclauses)
if news subsume clause then olds = olds less clause;;
   end ∀clause; newclauses = news; oldclauses = olds; return;
definef superfluous(clause); simplify external olds,news;
if olds subsume clause or news subsume clause then return t;;
return ∃t ε hd clause | tetl clause; end superfluous;
end simplify;
definef rebuild(clause); map=nl;
(∀var ε [u:tehd clause u tl clause] varsof(t)) map(var)=newat;;
return clause subst map; end rebuild;
```

Having built up all the necessary formula-manipulating preliminaries, we now go on to describe various of the principal theorem proving methods which appear in the literature on the subject. As has been mentioned, methods will often differ among each other simply because the strategies which they use to generate new clauses are different. The first method we shall

describe, *binary resolution with set of support*, divides the set of clauses from which it aims to derive a contradiction into two sets: those derived from the axioms of its subject, and those derived by negation of the theorem whose 'proof by contradiction' is to be generated. It then forms pairwise resolvents in all possible ways, never matching two axioms however, since a contradiction can never be reached through successive resolutions of axioms alone.

Note however that it is essential for the completeness of this method that, in forming resolvents for two clauses c and d, we attempt when possible to identify two or more units of c (or of d) by making appropriate substitutions in the units of c; consideration of the simple contradictory set

$$a(x) \vee a(c)$$
$$b(x) \vee b(c)$$
$$\neg a(x) \vee \neg b(x)$$

of three clauses illustrates this last assertion. A clause c' produced from c by a substitution which causes two of the units of c to coalesce is called a *factor*, and the term of c' produced by this coalescence is called its *distinguished* term. Of course, when we form a factor we will wish to perform resolution on its distinguished units rather than on any other term, since to proceed otherwise would be to perform work that is bound to be repeated later.

The SETL program for binary resolution theorem proving is as follows (see J. Robinson, A Machine-oriented logic based on the resolution principle, JACM 12, Jan. 1965, pp. 23-41 for the source of this method).

```
definef binres(axioms,antitheorems); matchup external newer;
old = axioms; new = antitheorems;
(while new ne nl) newer=nl;
if t ε(matchup[old,new] /*the procedure matchup will generate new
 resolvents*/ u matchup[new,old] u matchup[new,new])then return t;;
old = old u new; simplify (newer,old); new=newer; end while;
/* if no new resolvents then */  return f; end binres;

definef matchup(ca,cb);
   (∀x ε tandmaps(hd ca, tl cb))<tmx, mapx> = x;
newres = <hd ca subst mapx less tmx u(hd cb subst mapx),
          tl cb subst mapx less tmx u(tl ca subst mapx)>;
if newres = <nl,nl> then return t;
   else newres in newer;;end ∀x; return f; end matchup;

definef tandmaps(neg,pos); /*finds all maps which lead to
   the elimination of one or more  unit  during a resolution;
   returns a set of pairs of form <unit eliminated,map> */
   /* first sequence all the units */
nseq = nl; (∀na ε neg) nseq(#nseq+1) = na;;
pseq = nl; (∀pb ε pos) pseq(#pseq+1) = pb;; tmps = nl;
(1 ≤ ∀j ≤ #nseq, 1≤k≤#pseq|(map is mogenu(nseq(j),pseq(k)))ne Ω)
tm = nseq(j) subst map;
pile =[{nseq(m),j<m≤#nseq} u {pseq(m), k<m≤#pseq}]subst map
                  less tm; work = <tm,pile,map>;
(while work ne nl) x from work; <tmx,plx,mpx> = x;
<tmx,mpx> in tmps; /*sequence the 'pile' of units*/ seq=nl;
(∀t ε plx) seq(#seq+1) = t;;
(1 ≤ ∀n ≤ #seq | (xtrmap is mogenu(tm,seq(n)) ne Ω)
tmnew = tm subst xtrmap;
pilenew =[{seq(k), n<k≤#seq}]subst xtrmap less tmnew;
mapnew =mpx compose xtrmap; <tmnew,pilenew,mapnew> in work;
end ∀n; end while;  end ∀j;  return tmps;  end tandmaps;
```

In the theorem proving algorithm just given, the formation of factors is inefficient, since factors of a given clause will be formed again and again. We can avoid this inefficiency by restructuring our algorithm somewhat, so as to save factors once formed; this leads to the *factor resolution* method presented below. In this algorithm, we associate, with each clause c, the set *factors*(c) of all its factors. Note also that when a factor of a clause c is produced, we consider the particular one of its units which resulted from the fusion of two units of c to be distinguished, and, when the factor is used to produce a resolvent, insist that this distinguished unit disappear in the resolvent. For this reason factors are saved as triples in the following formats:

<'pos',unit,clause> ,

if they contain a distinguished positive unit, where *unit* is that distinguished unit, and where as usual *clause* = <*posunits*,*negunits*> gives the set of all positive and negative units of the factor; similarly, factors containing a distinguished negative unit are represented as triples

<'neg',unit,clause> .

The SETL algorithm for factor resolution is as follows.

```
definef factres(axioms,antitheorems);
old=axioms; new=antitheorems;factors=nl;
makefacts[old];
(while new ne nl) makefacts[new]; newer=nl;
if t ε (matchfact[old,new] /* the procedure matchfact will
   generate new resolvents*/
u matchfact [new,old] u matchfact [new,new]) then return t;;
old = old u new; oldkeep = old; simplify(newer,old);
(∀x ε oldkeep) if x ε old then factors(x) = Ω;; end ∀x;
new=newer; end while; /*if no new resolvents then */ return f;
end factres;
```

```
definef matchfact(ca,cb); factres external newer, factors;
/* first match unfactored terms */
(∀na ε hd ca, pb ε tl cb|mogenu(na,pb) ne Ω)
 if (newres is resolvent (na,pb,ca,cb))  eq <nl,nl>
   then return t; else newres in newer;; end ∀na;
/* now match factored terms */
(∀fct ε factors(ca), pb ε tl cb|hd fct eq 'neg')
   <-,dist,caf> = fct;
if (mgu is mogenu(dist,pb)) eq Ω then continue;; /* else */


if (newres is resolvent(dist,pb,caf,cb)) eq <nl,nl> then return t;
else newres in newer;; end ∀fct;
(∀fct ε factors(cb), na ε hd ca|hd fct eq 'pos')
   <-,dist,cbf >= fct;
if(mgu is mogenu(dist,pb)) eq Ω then continue;; /* else */

if (newres is resolvent(na,dist,ca,cbf)) eq <nl,nl> then return t;
else newres in newer;;  end ∀ fct;
           /* if arrive at this point then */ return f;
end matchfact;

define makefacts(cl); /*builds up all factors of a clause*/
factres external factors;
factors(cl) = {<'neg',fct>, fct ε negfacts(cl)}
   u {<'pos',hd fct, <*z3> fct, <*z2>fct>,
   fct ε negfacts(<tl cl,hd cl>)};
return; end makefacts;

definef negfacts(cl); /*builds factors by identifying
   'negative' units */
<negu,posu>= cl; facts=nl; /*make sequence of 'negative units*/
seq = nl; (∀x ε negu) seq(#seq+1) = x;; work=nl;
(1 ≤ ∀j ≤ #seq, j<k ≤ #seq|(map is mogenu(seq(j),seq(k))ne Ω)
ut = seq(j) subst map;
```

```
pile = [{seq(ℓ), k < ℓ ≤ #seq}] subst map;
clprime = <[hd cℓ] subst map, [tℓ cℓ] subst map>;
<ut, pile, clprime> in work; end ∀j;
(while work ne nℓ) x from work; <ut,pile,clprime> = x;
<ut,clprime> in facts; /*sequence the 'pile' of units */
seq = nℓ; (∀x ε pile) seq(#seq+1) = x;;
(1 ≤ ∀j ≤ #seq | (map is mogenu(ut,seq(j)) ne Ω)
ut = ut subst map; pil = [{seq(k), j<k<#seq}] subst map;
clprim = <[hd clprime] subst map, [tl clprime] subst map>;
<ut, pil, clprim> in work; end ∀j; end while;
return facts;
end negfacts;
```

Clauses may be distinguished as *positive* (containing positive clauses only) and *mixed* (containing both negative and positive clauses, or negative clauses only). The clause types which can result on forming the resolvent of a pair of clauses are as follows:

positive with mixed: null, positive, or mixed
mixed with mixed: mixed .

Note in particular that a null clause, i.e., a visible contradiction, can never result from two mixed clauses, but only from a positive and a negative clause. This remark suggests that positive clauses deserve special attention. Next observe that the process of resolution has a kind of 'associativity' property. If the resolvent c of two mixed clauses ca and cb is produced, and the resolvent d of c with a positive clause p is produced, then the term entering into the second resolution must be derived from a term either of ca or of cb, say ca for the sake of definiteness. We could therefore obtain the same resolvent by resolving ca with p, and then resolving cb with the result. Now, if d is positive (or null) it follows that the resolvent of ca with p must be positive. Arguing inductively, we see that every positive (or null) clause can be produced by a chain of resolutions such that one of the two clauses entering into each

resolution is positive. Thus, in searching for a contradiction, we may restrict ourselves to forming resolvents from pairs of clauses, one of which is positive.

This last remark leads to yet another observation. One principal difficulty which any resolution theorem prover must attempt to overcome is the tendency for clauses to accumulate rapidly during the functioning of the algorithm, leading eventually to swamping of available memory. Given that we may proceed by resolving positive with mixed clauses exclusively, it is clear that, if we are willing to resolve a mixed clause against an entire sequence of positive clauses, we need not bother to accumulate new mixed clauses, but can accumulate positive clauses only. If no new positive clauses can be accumulated, it follows that the set of clauses with which our algorithm is working is consistent. Resolution theorem proving by the use of this strategy was introduced by J. Robinson, c.f. Automatic deduction with hyper-resolution, Int. J. Comp. Math. 1 (1965) pp. 227-234, and is called *hyperresolution*. Note also that if a new positive clause can result by successive resolutions of a sequence of positive clauses with a given mixed clause m, it follows that all the negative terms of m must eventually be eliminated. Thus we may attempt to match the negative terms of m in any convenient order.

The following SETL algorithm embodies the theorem-proving strategy just outlined; note that the preceding routine *negfacts* is used, and that this returns a set of pairs

<nt,clause>

in which *nt* is a distinguished negative unit, and *clause* is a clause containing this unit and obtained by factorization.

```
definef hyperres(axioms,antitheorems);
all=axioms u antitheorems; setpos={cc all|hd c eq nl};
setmixed = all lesset setpos;
try:  (∀mc∈setmixed) do makepos(mc,setpos,newpos);
   if newpos ne nl then setpos=setpos u newpos; go to try;;
   end ∀mc;
/* if can form no new positive clauses then */ return f;
   block makepos(mc,setpos,newpos); work={mc};
   (while work ne nl doing work=rebuild[nework];)nework=nl;
(∀mcl ε work) /*first form negative factors and match
   against positive clauses*/  nfacts = negfacts(mcl);
(∀nf ε nfacts, poscl ε setpos, postεtl poscl
   |mogenu(hd nf, post) ne Ω) <dist,cl> = nf;
if (newres is resolvent(dist,post,cl,poscl)) eq <nl,nl>
   then return t;
else if (hd newres)  eq nl then newres in newpos
   else newres in nework;; end
/* next treat factors of positive clauses */ negt=  hd mcl;
(∀poscl ε setpos) pfacts=negfacts(<tl poscl, nl>);
(∀pf ε pfacts, nt ε negt|mogenu(negt,hd pf) ne Ω)
   <dist, ptms,-> = pf; pcl = <nl, ptms>;
if(newres is resolvent       dist,mcl,pcl)) eq <nl,nl>
   then return t;
      else if (hd newres) eq nl then newres in newpos
         else newres in nework;; end ∀pf; end ∀poscl;
/* now process unfactored clauses */
(∀poscl ε setpos, post ε tl poscl,nt∈negt|mogenu(post,nt)ne Ω)
if (newres is resolvent  (n:, post,mcl,poscl)) eq <nl,nl>
   then return t;
else if (hd newres) eq nl then newres in newpos
   else newres in nework;end if;
end ∀poscl; end ∀mcl; end while;
simplify (newpos,setpos); end makepos;
end hyperres;
```

Resolution theorem provers will often exhibit a kind of indecisiveness, forming a promising clause from a given clause c but then not pursuing the line of investigation opened up. Our next method, *maximal clash resolution*, seeks to overcome this difficulty by performing resolution repeatedly on the terms of a given clause as often as possible before a new clause is taken up. (Cf. J. Robinson, A review of automatic theorem proving, Proc. Symposia in Appl. Math. (1966), v. 19, American Mathematical Society, Providence, R. I. for the source of this approach.) In the SETL algorithms which follow, the following convention is used to facilitate the processes that must be applied. As resolution is repeatedly applied to the terms derived from an original clause *cl*, the clauses which result are maintained as triples in the form

<<ntmscl,ptmscl>,extneg,extpos> ,

Here *ntmscl* are all the negative terms, and *ptmscl* all the positive terms, which derive from terms originally present in *cl*; while *extneg* are all those other negative terms, and *extpos* all those other positive terms, which derive from clauses against which *cl* has been matched for the forming of resolvents.

The SETL algorithm for maximal clash resolution is as follows:

```
definef maxclash(axioms,antitheorems);
new = axioms u antitheorems; old=new; proved=f;
(while new ne nl) newer=maxresols[new];
if proved /*proved is a flag set by maxresols*/ then return t;;
old=old u new; simplify (newer,old); new=newer; end while;
/* if no new resolvents then*/  return f; end maxclash;

definef maxresols(clause);,maxclash external old, proved, newer;
if proved then return nl;; /*bypass work if finished already*/
work = {<clause,<nl,nl>>}; maxs=nl;
(while work ne nl) elt from work;
<<ntmscl,ptmscl>,xtrneg,xtrpos> = elt; canextend=f;
```

```
vars = [u: x ε ntmscl u ptmscl u xtrneg u xtrpos]varsof(x);
map = nl; (∀v ε vars) map(v) = newat;;
ntmscl = substin ntmscl; ptmscl = substin ptmscl;
   xtrneg = substin xtrneg; xtrpos = substin xtrpos;
extnegcl(ntmscl,ptmscl,xtrneg,xtrpos,'neg');
extnegcl(ptmscl,ntmscl,xtrpos,xtrneg,'pos');
if canextend then continue;; /* else */
newresol is <ntmscl u xtrneg,ptmscl u xtrpos> in maxs;
if newresol eq <nl,nl> then proved=t; return nl;; end while;
return maxs;                    end maxresols;

/*auxiliary substitution routine*/
definef substin x; maxresols external  map; return [x]subst map;
end substin;

define extnegcl(ntmscl,ptmscl,extneg,extpos,switch);
/* extends clash with negative terms of clause */
maxresols external map,work,canextend; maxclash external old;
(∀oldclause ε old)<ont,opt>=oldcl; oldpart=if switch eq 'neg'
     then opt else ont;
(∀x ε tandmaps (ntmscl, oldpart))
/* tandmaps will produce a set of pairs <term,map>, each
   showing a map which may be used to produce a resolvent
   and the term which resolution will then eliminate;the code
   for tandmaps has been given above */
            <term,map> = x; ntmsc=substin ntmscl;
ptmsc = substin ptmscl; extng = substin extneg;
   extp = substin extpos; ontl = substin ont;
optl = substin opt; ntmsc = ntmsc less term;

extng = extng less term u ontl lesset ntmsc;
extp = optl less term u extp lesset ptmsc;
if n ∃at ε ntmsc u extng|ute(ptmsc u extp) then canextend=t;
<<ntmsc,ptmsc>,extng,extp> in work; end if; end ∀x; end ∀oldcl;
return; end extnegcl;
```

The partial "associativity' of the resolvent-forming process, to which we have already alluded in our discussion of hyper-resolution, can be used in various ways to standardize any tree T of resolutions which leads to a contradiction, thereby cutting down on the number of alternatives which a theorem-prover algorithm must explore. In particular, suppose that a contradictory set s of clauses is given, with a subset n such that s-n is consistent. Then, by using the above-mentioned partial 'associativity' to force T into a pattern in which clauses are resolved against clauses of s (input clauses) as much as possible, it may be shown that we can always assume T to have a special form, which we may describe as follows. (Cf. R. Anderson and W. Bledsoe, A linear format for resolution with merging and a new technique for establishing completeness, J. A.C.M. 17 (3), 1970, pp. 525-534 for a proof).

Given two clauses c_1 and c_2 containing unifiable terms t_1, t_2, and given the map m which unifies these terms and the resolvent c produced from this unification, we call a term t of c a *merge term* if it arises both as $\bar{t}_1$ subst map and $\bar{t}_2$ subst map, where $\bar{t}_1$ is a term of c_1 and $\bar{t}_2$ is a term of c_2. Then, in the situation described above, T may be taken to consist of a sequence of resolutions, the i+1'st resolution producing a clause R_{i+1} from two clauses R_i and C_i, and where the following assertions hold:

i. Each C_i is either in s (i.e., is one of an original 'input' set of propositions) or is an R_j with $j < i$;

ii. If C_i is not in s, then C_i contains a merge term t, and the term resolved upon to produce C_{i+1} from R_i and C_i is t. Moreover, every term of R_{i+1} is a term of an appropriately

substituted version of R_i (subsumption condition).

iii. The initial clause R_0 belongs to T.

The assertions made in the preceding paragraphs establish the completeness of the modified resolution procedure, which may be called the *linear resolution* method, embodied in the SETL algorithm below. Note that during the i-th iteration of this algorithm (we refer here to the outermost 'while' loop) the set *curves*(c) represents all the resolvents R which might appear in the i-th place in a sequence of resolvents $R_0,\ldots,R_i$ satisfying the above conditions and with R_0 = c; except that if R would appear with lower index in such a sequence we drop it from *curves*(c) The set *offens*(c) represents all the resolvents R which could ever appear in such a sequence, only clauses which are properly subsumed by clauses subsequently appearing in *curves*(c) being dropped. The set *mergeresfrom*(c) represents the cumulative total of all resolvents belonging to such a sequence and containing a merge term.

The routine *matchup2*, modeled after the *matchup* routine used in the binary resolution algorithm given above, not only forms additional resolvents but notes those resolvents which are merge resolvents; merge resolvents are represented as triples <<negmerge,posmerge>,c>, where c is a clause containing merge terms, *negmerge* is the set of the negative merge terms which c contains, and *posmerge* is the set of positive merge terms which c contains. The routine ca subsinst cb tests to see whether there exists a *map* such that every term of ca is included among the terms of cb subst *map*. The technique is rather like that used in the subsumes routine given earlier: All the variables of ca are redesignated as constants; a set of *maps* is initialized to consist of the identity map only. Then, proceeding through the terms t of ca successively, and for each *map* ε *maps*, one tests to see if *map* can be extended to a *map*'in such a way as to make t agree

with a term of cb subst map'. If a sequence of extensions ultimately covering every term t of ca can be made, *true* is returned; otherwise *false*.

Here is the SETL algorithm.

```
definef linres(axioms,antitheorems);
input=axioms u antitheorems;
curres = {<c,{c}>, c∈antitheorems}; allres = curres;
mergeresfrom = {<c,nl>, c∈antitheorems};
(while ∃c∈antitheorems | curres(c) ne nl)
/* as long as there are current resolvents*/
(∀origcl ∈ antitheorems) newres=nl;
(∀ cl ∈ curres(origcl))<neg,pos>=cl;
(∀item ∈ mergeresfrom(origcl))<<negmrg,posmrg>,clm>= item;
   if matchup2(neg,posmrg,cl,clm) or matchup2(negmrg,pos,clm,cl)
         then return t;; end ∀item;
clm = Ω; /*to control a detail of the processing by matchup2;
note that matchup2 will define a new total set of resolvents,
   and a new set of merge resolvents*/
(∀clin ∈ input)<negin,posin> = clin;
   if matchup2(neg,posin,cl,clin) or matchup2(negin,pos,clin,cl)
         then return t;; end ∀clin;
end ∀cl; nowres=allres(origcl); simplify(newer,nowres);
allres(origcl)=nowres u newer; curves(origcl)=newer;
end ∀origcl; end while; /*if no surviving resolvents then*/
return f; end linres;

definef matchup2(neg,pos,ca,cb);
/*forms resolvents and notes merge resolvents */
/* note use of subroutine tandmaps  given previously*/
linres external mergesfrom,newer, cl,origcl,clm;
(∀x ∈ tandmaps(neg,pos))<tmx,mpx>=x;
newres = <nl is(hd ca subst mapx less tmx))
   u (n2 is(hd cb subst mapx)),
   pl is (tl cb subst mapx less tmx) u (p2 is tl ca subst mapx))>;
```

```
if newres eq<nl,nl> then return t;;
if n clm eq  Ω/* so that not matching an input clause
   but an old merge resolvent*/  then if
    n (newres subsinst clm) then continue;; end if; /*else*/
newres in newer;
mergeparts = <n1 int n2, p1 int p2>;
if mergeparts ne <nl,nl> then
   mergeresfrom(origcl)=mergresfrom(origcl)
   with <mergeparts,newres>;
end ∀x; return f; end matchup2;

definef ca subsinst cb; /*determines whether there exists a
   map such that every term of ca is included among the terms
   of cb subst map */
varsa = [u: t ε hd ca u tl ca] varsof(t);
makconst = {<x,{x}>, x ε varsa};
<nega,posa> = <[hd ca] subst makconst, [tl ca] subst makconst>;
maps = {nl}; /*start with set consisting of identity map only*/
<negb, posb> = cb;
(∀ta ε nega) if (maps is extend2(maps,ta,negb)) eq nl
   then return f;;;
(∀tb ε posa) if (maps is extend2(maps,tb,posb)) eq nl
   then return f;;;
/* if arrive at this point then */  return t;
definef extend2(maps,term,termset); /*auxiliary routine
   which tests to see if by extending some map ε maps to a
   newmap, term can be identified with an element of termset
   subst map subst newmap*/
extensions = nl; (∀m ε maps, t ε termset)
if(addition is mogenu(t subst m, term)) ne Ω
   then addition u m in extensions;; end ∀m; return extensions;
end extend2; end subsinst;
```

D. Loveland has given a theorem-proving algorithm rather closely related to the linear resolution method discussed just above; Loveland calls his procedure *model elimination*. (See Loveland, A simplified format for the model elimination theorem-proving procedure, J. ACM v. 16, pp. 349-363 (1969)). Like the linear resolution method, model elimination forms resolvents using input clauses whenever possible. The 'merge resolvent' case arising in the preceding algorithm is handled somewhat differently, however. The technique used is as follows. As resolvents are formed a partial record of their parentage is kept. Resolvents are in fact maintained as sequences $t_1,\ldots,t_n$ of units (Loveland calls these sequences 'chains'). Certain of the units in a sequence are specially flagged. A sequence corresponds, in the more customary resolution theorem provers, to a clause formed by a succession of resolutions; flagged units are those which resolution would have eliminated. In Loveland's procedure, these units are kept (but specially marked) so as to permit the subsequent elimination of additional units by a logical equivalent of the 'factoring' process applied in conventional resolution theorem provers. It is not hard to see that if resolvents are kept in this form the following three processes must be applied:

1. Extension (Corresponding to simple resolution). Take a sequence $t_1,\ldots,t_n$ whose last unit is unflagged. Find an input clause c and a unit t in c, having negativity opposite to that of t_n, and such that t and t_n can be unified by a map m. Apply the map m to all the terms of $t_1,\ldots,t_n$ and to all the terms of c, thus obtaining a new sequence $t_1',\ldots,t_n'$ and a new clause c'. Eliminate t map m from c' and append the remaining units of c to $t_1',\ldots,t_n'$; flag t_n'.

2. Reduction (Corresponding to factoring). If a sequence $t_1,\ldots,t_n$ contains a marked unit t_j which is followed by an unmarked unit t_k of oppositive negativity, and if there exists

a map m which unifies t_j and t_k, then apply m to all the units of $t_1,\ldots,t_n$, and drop the k-th unit of the result.

3. Contraction (Dropping terms no longer needed). If a sequence ends with a marked unit, drop this unit.

Process 3 should be applied when possible; if this is not possible, then process 2; otherwise process 1. A contradiction is represented by a sequence of length zero. A sequence containing two marked identical units of the same negativity represents a resolvent into whose formation a tautologous clause has entered; such a sequence may therefore be ignored. A sequence containing two marked identical units of opposite negativity represents a clause subsumed by a shorter clause, and may therefore be ignored also. Finally, a sequence containing two unmarked identical terms of opposite negativity is a tautology, and may be ignored for this reason.

In the following SETL algorithm, the 'sequences' spoken of above will be represented as sequences of triples

<unit,posneg,mark>

where *posneg* is t if the unit is to be considered as unnegated, f otherwise; *mark* is t if the unit is to be considered as marked, f otherwise.

```
definef modlim(axioms,antitheorems);
all = axioms u antitheorems; chains = nl;
(∀x ε antitheorems) seq =  nl; <neg,pos> = rebuild(x);
(∀nu ε neg) seq(#seq+1) = <nu,f,f>;;
(∀pu ε pos) seq(#seq+1) = <pu,t,f>;;
seq in chains; end ∀x;
(while chains ne nl) /* first apply reduction process to
   produce additional chains */  redchains=chains;
(while redchains ne nl) newred = nl;
(∀seq ε redchains, 1≤j<#seq, j<k≤#seq |
 <*z3>seq(j) and n <*z3>seq(k) and <*z2> seq(j)ne<*z2>seq(k)
 and (map is mogenu(hd seq(j), hd seq(k))) ne Ω)
newseq = {<hd seq(ℓ) subst map, tl seq(ℓ)>, 1≤ℓ<j};
utm = hd seq(j) subst map;
(j < ∀ℓ ≤ #seq) utlm = hd seq(ℓ) subst map;
if utlm eq utm and n <*z3> seq(ℓ) and <*z2> seq(j)ne<*z2>seq(ℓ)
   then continue;;
/*else*/ newseq(#newseq+1) =<utlm,tl seq(ℓ)>; end ∀ℓ;
if trim(newseq) then return t;; /*else*/
   if n ignore then newseq in chains; newseq in newred;; end ∀seq;
redchains = newred; end while;
/* now apply extension process */  newchains=nl;
(∀seq ε chains, incl ε all) if <*z2> seq(#seq) then go to poslast;;
/* else negative unit is last */
(∀pu ε tl incl | (map is mogenu(pu, hd seq(#seq))) ne Ω)
<mapneg,mapos> = <[hd incl] subst map, [tl incl] subst map
   less (pu) subst map)>;  do buildnewseq; block buildnewseq;
newseq = {<j, hd seq(j) subst map, tl seq(j)>, 1≤j≤#seq};
 newseq(#seq) =<t r 3> newseq(#seq); /*'mark' this unit */
(∀ut ε mapneg) newseq(#newseq+1) = <ut,f,f>;;
(∀ut ε mapos) newseq(#newseq+1) = <ut,t,f>;;
```

```
if trim(newseq) then return t; else if n ignore
   then newseq in newchains;; end buildnewseq;
end ∀pu; /* completes treatment of trailing negative unit;
   treatment of trailing positive unit follows */
[poslast:]
   (∀nu ε hd incl | (map is mogenu(nu,hd seq(#seq))) ne Ω)
   <mapneg,mapos> = <[hd incl] subst map less (nu subst map),
                    [tl incl] subst map>;
   do buildnewseq;
   end ∀nu;
   end ∀seq; chains = newchains, end while;
   /* if reach this point then */  return f;
   end modlim;

definef trim(seq); modlim external ignore;
(while <*z3> seq(#seq)) seq(#seq) = Ω;
if seq eq nl then return t; end while;
/* now check for various conditions of redundancy */
ignore = f;
if 1 ≤ ∃j < #seq,  j < k ≤ #seq |
   <*z3> seq(j) and <*z3> seq(k) and hd seq(j) eq hd seq(k)
$or n <*z3> seq(j) and n <*z3> seq(k) and hd seq(j) eq hd seq(k)
   and <*z2> seq(j) ne <*z2> seq(k) then ignore = t;
return f;end if;
/*if not redundant, then substitute new variables before return*/
vars = [u : 1 ≤ j ≤ #seq] varsof(hd seq(j)); map = nl;
(∀v ε vars) map(v) = newat;;
seq = {<j,hd seq(j) subst map, tl seq(j)>, 1≤j≤#seq};
return f; end trim;
```