AD-773 839

A GENERATIVE, NESTED SEQUENTIAL BASIS FOR GENERAL PURPOSE PROGRAMMING LANGUAGES

CARNEGIE-MELLON UNIVERSITY

PREPARED FOR
AIR FORCE OFFICE OF SCIENTIFIC RESEARCH
ADVANCED RESEARCH PROJECTS AGENCY

NOVEMBER 1973

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR - TR - 74 - 0097	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER AD 773839
4. TITLE (and Subtitle) A GENERATIVE, NESTED-SEQUENTIAL BASIS FOR GENERAL PURPOSE PROGRAMMING LANGUAGES		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) David Sheridan Wile		8. CONTRACT OR GRANT NUMBER(s) F44620-73-C-0074
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Department of Computer Science Pittsburgh, Pennsylvania 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101D AO 2466
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209		12. REPORT DATE November 1973
		13. NUMBER OF PAGES ~~159~~ 161
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Air Force Office of Scientific Research (NM) 1400 Wilson Blvd Arlington, Virginia 22209		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This research effort presents a new approach to programming language design. Essentially, we have studied the close relationships of program structures and the XXXXX data structures they use, and found that reorienting programming style to emphasize these relationships is a fruitful direction for future language designs. As a vehicle for studying these relationships, a language "basis" is developed--a set of primitives, a syntax, and an interpretation. The basis is "incomplete" in that it does not define a real programming language, nor are all "fundamental" aspects of language design considered. However, one is able

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Block 20. Abstract (Continued)

to describe many algorithms concisely in the basis, which leads us to believe it represents a significant step in programming language design. XXXX Our approach to the design of the basis may be characterized as a derivative of "structured programming" studies. In particular, "gotoless programming" proponents advocate replacement of most explicit program pointers ("goto's) in programming languages by a set of control construct--grouping, subroutine call, conditional, selection, looping, and escape facilities--which impose a nested-sequential statis structure on programs. The germinal idea of this work is that perhaps the gotoless constructs can be applied to nested-sequential structures in general--independent of whether the structure is thought to represent program or data. We will then have a "pointerless" representation for both structures.

A GENERATIVE, NESTED-SEQUENTIAL BASIS FOR GENERAL PURPOSE PROGRAMMING LANGUAGES .

David Sheridan Wile

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

November, 1973

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-73-C-0074) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.

ib

ABSTRACT

This research effort presents a new approach to programming language design. Essentially, we have studied the close relationships of program structures and the data structures they use, and found that reorienting programming style to emphasize these relationships is a fruitful direction for future language designs. As a vehicle for studying these relationships, a language "basis" is developed--a set of primitives, a syntax, and an interpretation. The basis is "incomplete" in that it does not define a real programming language, nor are all "fundamental" aspects of language design considered. However, one is able to describe many algorithms concisely in the basis, which leads us to believe it represents a significant step in programming language design.

We may ascribe the primary influences on the work to structured programming studies and the programming languages Bliss, APL, and LISP. Secondary influences include formal approaches to language design and program optimization. The parallel work of Backus shares the basic precepts of pointerless representation and the concise nature of combinatoric constructs; however, they differ radically in both approach and emphasis.

Our approach to the design of the basis may be characterized as a derivative of "structured programming" studies. In particular, "gotoless programming" proponents advocate replacement of most explicit program pointers ("goto"s) in programming languages by a set of control constructs--grouping, subroutine call, conditional, selection, looping, and escape facilities--which impose a nested-sequential static structure on programs. The germinal idea of this work is that perhaps the gotoless constructs can be applied to nested-sequential structures in general--independent of whether the structure is thought to represent program or data. We will then have a "pointerless" representation for both structures.

Another desirable aspect of programs is that the invocation of the "next" instruction to be executed is implicit; conventional data structures, on the other hand, must be explicitly "pulsed" to obtain the next element. Very often, a one-to-one identification can be made between program elements and the data structure elements they access. (For example, a one-to-one identification between elements of an array and the incarnations of a loop body can frequently be made.) The basis is designed to emphasize this aspect of programs through the use of operators which apply programs to data "cosequentially".

One of the "gotoless" constructs which does constitute use of an explicit "program pointer" is the (potentially recursive) subroutine call. The research proposes structuring the use of this pointer by substituting a set of "recursionless" constructs in traditional programming languages (such as Algol); the analogy with "gotoless" constructs is direct--no explicit recursive calls are required for recursive effects. The data structure involved in implementing a recursive function parameter mechanism follows the control structure so explicitly that a stack is frequently used to contain both data and control information. "Corecursive" operators--directly analogous to cosequential operators--are thus introduced to emphasize the close relationship between recursive data and control structures.

In specifying a programming language "basis", we are admitting open-endedness a priori. We feel that a formalization of this basis should prove beneficial to program correctness techniques and formal semantics specification language development. Additionally, program/data structure optimization and representation issues are unified by the approach. We expect that a language developed from this basis will be analogous in power for nested-sequential structures to APL for homogeneous, parallel structures.

ACKNOWLEDGEMENTS

I wish to express my gratitude to my thesis advisor, Professor William Wulf, for his constant encouragement, direction and restraint. I would also like to thank the members of my thesis committee, Professors Raj Reddy and Andrew Wong, and especially Professor Mary Shaw for her superb nitpicking.

My sincere thanks to my teachers, professors, friends and family for their guidance, and especially to my wife, Linnea, for her patience, support and encouragement during my long career as a graduate student.

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION AND APPROACH

Goals

All research in the field of programming language design requires justification in this era of language proliferation. The fashionable criterion for effective progress in this field is that any new language provide an "order of magnitude improvement" over the existing paragon. Until a language meets the criterion, there should be no new compilers, no reprogramming, no low-yield design efforts, etc. Aside from the obvious fiscal benefits, new language projects should decrease and the field of programming languages should advance by more fundamental, large increments. Additionally, the criterion is "suitably vague": a designer who claims his new language meets it must identify the nature of the improvement that the language represents.

The primary goal of this dissertation is to indicate that an order of magnitude improvement in general purpose programming languages is possible, and to provide a basis for such a language. The basis is such that its extension to a real programming language is non-trivial. Rarely can traditional language constructs enter the language unaltered; frequently, they are found already embedded in the language in a fundamental way. Some justification will be given for suspecting that consistent incorporation of truly alien constructs into the language tends to be "synergistic"--the actual gain is more than could have been expected from experience with the constructs in other languages.

Indeed, the development of the language basis did not proceed from the criterion above, but rather from insight into a pointerless program/data structure representation arising from structured programming "gotoless" program studies. The resultant basis was observably more concise than general purpose programming languages such as Algol [NA]. Hence, as an attempt to discern the source of this conciseness, the order of magnitude criterion was studied in terms of languages which have in some sense met the criterion (with respect to their predecessors). We concluded that fundamental improvement in languages has not arisen from extension of contemporary languages, but rather by a reformulation of languages which emphasize common interrelationships of concepts obscured in their predecessors.

The somewhat pretentious claim that our considerations of the germinal "pointerless representation" concept may lead to a programming language which meets the order of magnitude criterion is not intended to belittle the original considerations in the development of the basis, but rather to emphasize their importance.

We proceed by placing this work in the programming language design milieu. Next the order of magnitude improvement criterion is examined more closely, along with aspects of language design contributing toward its satisfaction. We then establish the approach and essential concepts embodied in the remainder of this work.

Programming Language Design: State of the Art

Most higher level languages of the early 1960s did meet the "order of magnitude" criterion when compared with machine language or even, in some cases, FORTRAN (e.g. APL, SNOBOL and LISP for wide (disjoint) classes of problems)†. However, in the late 1960s and the early 1970s computer scientists began to focus on the fundamental concepts underlying the activity of programming and the machines for which programs are written.

1. The stimulation of machine technology

The rapid pace of hardware innovations has certainly kept one group of programming language designers active; machines such as the Star [HT] and ILLIAC IV [BN] have features to occupy designers in merely allowing the higher-level language programmer to use the machine effectively. A related group of designers--disenchanted with the inefficiencies of general purpose languages--have resorted to lower-level machine-oriented implementation languages; languages such as Pascal [WI] and Bliss [WU,1971] have in fact contributed to the field of higher-level language design, although that was at most a subordinate goal in their design.

The technology has further stimulated programming efforts in systems concepts; paging, networks, associative memories, protection systems, etc., have all caused existing programming languages to be reexamined, especially to determine whether applicability of the concepts should be discovered by compilers or extensions made to facilitate their use directly. Real-time facilities, exotic new peripherals, "applications" systems, and microprogramming are only beginning to influence programming language design--as evidenced by recent interest in "two-dimensional languages", for example [WM]. Despite activity in the technological areas, with the exception of implementation languages and graphics languages, the approach has been to extend existing languages, rather than to invent new languages incorporating or anticipating hardware technology in any fundamental way. It is more probable that we do not understand the implications of features like streaming and parallelism than that they will not ultimately affect the

- -

†See [BA,1957], [McCR], [IV], [PAK], [FGP], [GPP], and [McC].

heart of programming language design.

2. Formal and informal methodologies

Another group of programming language designers has begun to address the problem of finding the fundamental underlying concepts of the activity of programming. There are many approaches to this problem. Formal approaches include formal semantics specifications for entire programming languages, program "schemata" studies, program verification efforts, and various axiomatizations suitable for correctness proofs for particular programs in specific languages†. Rigorous approaches to systems programming problems--cooperating sequential processes and protection schemes, for example--provide practical problems for which formal analysis is a tool of obvious and immediate benefit††.

"Structured programming" presents a more empirical view of the activity of programming; here the language design issue is primarily "can we use what we have?" and only secondarily "how should we improve it?"†††. Even here the approach tends to be mildly formal--properties preserved or destroyed by control constructs, for example, are closely examined and the "correct" way to program is preferably the way which may be proved correct. However, the formal approach is a means to an end: enhancement of understandability, principally through enforcement of a hierarchical programming style. Considerable practical experience has led theoretical computer scientists to accept "gotoless programming" as an improved technique [DI,1968], [WU,1971]. Also, features for controlling the ill-structured properties of global variables are emerging [WS].

A related issue is the management of large programming efforts--modularization **a la** Parnas [PA]. Conventional decompositions of programming tasks tend to maximize knowledge of the interfaces between components; a modularization which minimizes such knowledge has been found to result in more easily modifiable systems. Unfortunately, the decomposition is often orthogonal to that proposed by "structured programming" enthusiasts.

- -

†See [AJS], [IA], [CG], [SN], [KI] and [GR].

††See [DI,May,1968], [HA], and [BH].

†††See [DI,1969], [HO], [WU], and [WS].

Other management issues are attacked by those interested in bootstrapping and transportability†; the major impact of these issues on programming language design is to cause more exact distinction between the fundamental features of a language and those which are actually syntactically or interpretively extensible from the language core or present for efficiency. Those interested in program management are approaching language design from a fundamentalist viewpoint; we may expect at least the core of future languages from this group to be very sparse, compact and logically consistent.

3. Maturity

Of course, it is unfair to divide programming language design exclusively into two camps--technology and management. General purpose languages continue to be designed and implemented. However, many computer scientists feel that large "omnibus" languages such as Algol 68, PL/1 and Simula 67†† represent the end of the large language era for several reasons:

1. They have stimulated enough problems in their implementation and description to keep computer scientists busy simply trying to understand them;

2. Experience with them is so limited that no one can propose absolutely better solutions for the problems they pose; in particular, meager evidence does not show an "order of magnitude" return for the investment, either in size or complexity decrease;

   and

3. There is a general hope among computer scientists that programming languages need not be that complex.

Programming language design has also matured significantly since the early 1960s. Algol 60 [NA] is no longer thought divine, but has rather entered the small group of universally understood languages--along with FORTRAN, LISP, and SNOBOL. APL is generally taken more seriously than previously, although its merits are far from

---

†See [BR], [WA,1967,1970].

††See [vW,1969], [IBM], and [DMN].

universally recognized. Higher level languages have become viable alternatives to assembly languages for systems problems.

Emphasis in the language design field has shifted from the study of syntax and compiler-compilers to "semantics"; the former studies have been subjugated to the study of extensible languages, pursued by an active (though currently disillusioned) group of designers [SCH]. Issues of scope, storage management, control structures and data structures--although in no sense resolved--are considerably better understood.

The new languages which have arisen from hardware technology or program management studies have been treated in the literature with emphasis on the technical or management issues they consider.

In summary, those computer scientists in the mainstream of programming language research are not designing programming languages.

Language Design: Trade between Technology and Formalism

General purpose programming language designers tend to emphasize either formal or technological innovations in the languages they design; however, they cannot satisfactorily rely wholly on either. The disparity between machine/system design and formal axiomatization makes formal languages unusable from a practical standpoint. Similarly, the often **ad hoc** nature of advances in machine design, which are frequently poorly matched with the systems and languages in which they are ultimately embedded, makes a purely technological approach unacceptable from a management viewpoint.

Certainly, a trend of the past decade is toward mutual trade between the formal and technological fields. In particular, formalists no longer propose alternatives to Turing Machines unless the emphasis is toward a more practical, realistic model of computing; i.e., the emphasis is no longer a basis for computablity. Computers, not computability, motivate the formal approaches to problems such as assignment, data structure axiomatizations, and (to some extent) complexity. Analogously, programming languages have borrowed the mathematically precise notions embodied in association mechanisms and set operations. "Structured programming" and systems programming algorithms are frequently proved correct, using rigorous approaches previously found only in the more formal studies.

The language basis developed below borrows extensively from both progamming languages and formal applications. In searching for a powerful computation base, applicative languages are found to provide a pointer-free representation for computation. Without destroying this representation by introducing the troublesome notions of "assignment" and "side-effects", a sequence generation mechanism is included as an abstraction of the process which computers perform. Although the notations used appear formal, the treatment is not. The results are presented as fruitful directions for the design of future programming languages, not as an attempt to incorporate particular aspects of current hardware technology into a formal basis for computation.

Order of Magnitude Improvement

The "order of magnitude improvement" criterion is intentionally vague; many individual aspects of programming languages could be improved by a factor of ten without a corresponding improvement in the overall task of designing, writing, debugging, running, and modifying a program in the langauage. Possibly the only precise definition of an order of magnitude improvement in general purpose programming languages would be economic--in terms of the total cost of developing and using a program, measured for a diversified group of programs and programmers over an extended period of time†. Certainly programming in the improved language must become more natural to a large group of programmers, and implementations of programs must become more powerful.

Insisting that the implementation of a language be efficient on current machines requires the definition to be relatively insensitive to hardware advances. This poses two requirements: (1) the language must be futuristic enough to predict machine technology advances, lest it be obsolete immediately, and (2) it must not depend on future technology for its acceptance. However, it is unlikely that a language meeting the criterion could be developed which does not require at least some improvement in optimization techniques for current machines; specialized languages such as APL and SNOBOL which have (intuitively) met the criterion certainly require such advances for acceptable implementation efficiency. Hence, although we wish to constrain our considerations of the order of magnitude criterion to language design as influenced by a natural conceptualization of problems, implementation considerations cannot be ignored.

- -

†The criterion probably arose in response to the economic question: how much better must a language be to warrent the vast implementation and programmer retraining costs entailed by a new language? This initial overhead factor would be required in the definition.

Although we do not propose an exact definition of the order of magnitude improvement criterion, it is clear that one economic effect must be that programming must take less time than it does currently.

There are several approaches to decreasing the time required to program. Ignoring the special-purpose language approach (we want a "general purpose" language), the principal methods seem to hinge about the ability of a language to eliminate the specification of "detail" that is necessarily specified in other programming languages. Hence, for our purposes, we shall assume that a language in which programs need contain only one tenth of the detail that would be required in current languages, represents an order of magnitude improvement in general purpose language design. At least two constraints are appropriate: (1) the new language must be implementable approximately as efficiently as those against which it is compared, and (2) the gains in concise specification of algorithms must not suffer from the "write only language" syndrome--complexity of interpretation of language constructs must not defeat gains in conciseness. The principal effect of this latter constraint is to insure that programs are not necessarily poorly structured. Below we present a discussion of language design mechanisms for eliminating detail. Features will be presented as methods for introducing conciseness; aspects which mitigate against structure or efficiency will be mentioned.

1. **Conciseness**

Higher level language design may be viewed as an attempt to make programs more concise while preserving their structure. Note first that, although concise primitives are desirable, a blind attempt to minimize the number of primitives involves some loss of efficiency whenever the underlying model is more powerful than the language itself. For example, if the successor function is the only mechanism for addition, any implementation on a machine with addition as primitive will be inefficient.

One of the principal methods for reducing a program's length is to eliminate or limit the number and scope of "temporarily defined names" used by the programmer†. The most prevalent mechanisms for achieving this end are the inclusion of operators in a language and the ability to define functions. Both have the effect of making the program more concise by eliminating the necessity to initialize a temporary name--e.g. a register--and then perform an operation on it--e.g. a machine code command. The recent trend toward "expression languages" is an extension of this notion; languages

- - - - - - - - - - - - - - - - - - - -

†Traditionally the phrase "temporarily defined names" refers to internal names generated by a compiler. Here we mean names generated by the programmer for temporary use.

such as Gedanken, Euler, Bliss, Algol 68, etc.†, use the LISP notion of associating a value with each construct in the language--including those normally thought of as control and command constructs. Limitation of the accessibility of temporary names is the primary motivation for hierarchical scope rules in programming languages. Such rules lessen the bookkeeping required of the programmer by permitting identical names in different contexts. In addition, they provide a minimal concession to the preference of natural language for context dependent interpretation. Although the efficiency issues involved are by no means trivial or completely solved, most programmers prefer the use of these mechanisms with the slight loss of efficiency incurred.

Another method for reducing program size is to provide a large number of highly specialized primitives in the language; the conciseness of APL is due in part to this property of the language. Clearly, if for most programs one need define only one tenth as many functions in one programming language as in another, the former represents an order of magnitude improvement over the latter (for a fixed performance level), particularly if their invocation is concise as with APL single character operators. One can usually identify subsets of related primitives in languages where a large number are available--for example, subsets related to strings, arrays, boolean variables, etc. Efficiency problems do not generally arise when constrained to these subsets. However, some languages--PL/1 and Algol 68--allow implicit relationships among the sublanguages defined by the subsets: one may add a boolean to a real number. Although this is an aid to conciseness (explicit conversion calls are not necessary) it increases the complexity of interpretation, and may cause a loss of efficiency. Extensible language enthusiasts tend to downgrade this aspect of language design, preferring a minimal "kernel" from which all extensions are made. Although the "kernel" notion is excellent as an aid to the description and development of a large number of primitives, the lack of a large number of primitives in the language itself has severe implications to its implementation and its utility as a tool for communication.

A third method for gaining conciseness involves the reformulation of groups of concepts in languages in natural or structured ways despite a mismatch in their implementation. In some sense, each higher-level language construct causes a local loss of efficiency in implementation; those which are of most benefit frequently increase knowledge required of the relationships among language elements for efficient implementation even though the relationships are not recognized at the source language level. Current optimization techniques are concerned with discovering these relationships. The technology will be directed by such language efforts, although it has not been significantly prodded by them in the past; i.e., programming language design

- -

†See [RE,1970], [WW], [WU,1971], and [vW,1969].

tends to lag behind machines from fear of inefficient implementation on current machines. Sadly, natural formulations in programming languages of techniques such as streaming and microprogramming are not their raisons d'etre.

Fortunately, recent developments in structured programming (e.g. gotolessness) and operating systems (e.g. P and V synchronization) have emerged despite inefficiency qualms (control structures were subsequently optimized and now no one really worries about their inefficiency). Languages like SNOBOL and APL are particularly noteworthy in that they have provided a conceptual framework for constrained data structures. Their conciseness for problems over the class of data structures for which they were designed is remarkable; their power derives from the fact that detail related to implementation of the structures and operations on them is subsumed in the implementation. That is, enough is known about the structures that an acceptably efficient implementation can be built; the conciseness is often worth the price of even poor implementations. Of course, these languages (in particular) lose their leverage when problems outside their respective realms are attempted. Although inclusion of more sophisticated control constructs would have enhanced their applicability, the scope of their power is limited by the inability of their conceptual structures--strings and homogeneous arrays--to model many aspects of the structures used in computing. Naturally, part of their power derives from the assumptions they can make about these limitations.

2. Structure

Conciseness as an ultimate goal in language design has some limitations. Natural languages in particular contain redundancies which emphasize linguistic structure--which then render unnaturalness "apparent" as was mentioned above. Redundancy in programming languages is minimal, although one might argue that the preference of parenthesization to postfix notation in languages constitutes a concession to redundancy. We are unable to propose any particularly effective methods for utilizing redundancy in language design. However, a "structured decomposition" of a language, and of the specifications of algorithms in the language, is considered desirable.

Inasmuch as "conciseness" has received considerable emphasis as a means for meeting the order of magnitude criterion, its relationship to "structure" deserves some attention. Reactions to APL "one-liners"† often leads (non-APL) programmers to believe

- -

†The "one-liners" are normally extremely complex, involve several APL operators, "just fit" on one line, and accomplish tasks of considerable difficulty.

that the ultimate in conciseness is incomprehensibility. Indeed, encoding techniques more properly constitute the ultimate in conciseness, and may even have incomprehensibility as their goal.

However, we claim that most APL "one-liners" are not sufficiently concise! The principal grounds for such a claim lie in the fact that "structure" often aids conciseness to the extent that once inside the context of a structured entity, the representation for any particular effect is more concise than were the structure not present. For example, the principal reason the APL one-liner is concise in the first place arises from the vast structure of APL--in whose context the expression must be interpreted. The claim that most APL one liners are not sufficiently concise is simply a claim that their programmers have not found structural similarities between the one-liner and the other programs they have written. That is, in the context of programming (over a period of time, as opposed to writing a particular program), the one-liner is probably decomposable into previously obtained effects which should have been factored from the expression as functions. Certainly, when common effects can not conceivably be recognized--i.e., when generalization seems unlikely--we prefer a structured decomposition of a one-liner, while granting its superior conciseness.

In summary, we must find intrinsically powerful primitives whose relationships, though complex, provide sufficiently constrained assumptions for efficient implementation, and which are amenable to change as well as concise description. In the remainder of the thesis, a broader base for structures--nested sequences--is used. We can therefore expect our assumptions to be less powerful than either of the above languages allow. However the power gained by the naturalness of this structure to machine computation should, in general, offset that lost by the less rigid assumptions, and a more concise language for a wider class of problems will result.

Pointerless Representation

1. **"Gotoless programming"**

We begin by examining one recent technique developed by proponents of structured programming--removal of the goto statement from programming languages. Essentially, this removal involves the substitution of a "complete"† set of control constructs for the goto. For example, the set might include subroutine call, selection,

- -

†"Complete" in a pragmatic sense, more than mathematical. In particular, the set is in no way computationally "minimal".

grouping, conditional, loop and escape facilities. The set mentioned is not complete in the sense that every control structure obtainable with the goto can be simply represented using these constructs; rather the philosophy imposed is that the constructs point out those uses of the goto which are dangerously complex. In fact, the set mentioned is not complete from another viewpoint: the forced representation of certain useful, safe constructs is unduly complex. (The coroutine, "enable" mechanism, and "select" expression in Bliss represent concessions to this incompleteness [WU,1972].) Gotoless language designers resist the urge to reintroduce the goto in order to obtain such constructs, preferring to extend the set of gotoless constructs.

One of the principal benefits of gotoless languages arises from the effective removal of explicit program pointers from the language (by constraining pointers to particular objects in a local context). While the programs which can be written in the language are visibly less complex than in a language with gotos, the complexity of the underlying structure discoverable by a compiler is actually increased. This increase is not simply an amount which would make the efficiency of gotoless languages comensurate with languages with the goto, but is actually a significant increase beyond that, arising from the assumptions about language elements which gotolessness allows. Recent work in code optimization by Geschke [GE] exploits this gain; recent work by Hansen [HAN] indicates that such considerations need not lead to inefficiencies in the compilation process.

2. Pointers in programs

An examination of programming languages in general indicates that misuse of the goto (this notion is now well-defined) is only symptomatic of problems introduced by the use of pointers in programming languages. In effect, gotolessness controls program-to-program pointers; it says nothing about program-to-data and data-to-data pointers, both of which present problems at least as complex as those introduced by the goto. The problems incurred are analogous to those of the goto; "unnaturally complex" entities can be built easily by the programmer and consideration of these by the compiler writer defeats real gains which could be made if they were controlled.

For the unconvinced, brief examples of problems incurred with both of the above types of pointers may be enlightening. Program-to-data pointers--yes, "variables"--incur problems primarily from the side effects of the assignment operator. When a programmer calls a subroutine (procedure) he is often unaware of global variables which may change as a result of the call; common subexpression optimization facilities in a compiler are also thwarted by this phenomenon. In effect, both the compiler and the programmer must assume that any global variable may have changed†.

Scope mechanisms are inadequate here--they do not constrain the access to global variables effectively. (This problem is currently being studied independently as a "structured programming" problem [WS].)

It is a rare programmer who has not incurred problems with data-to-data pointers; LISP programmers using "nconc", "replaca", and "replacd", for example, create cyclic lists unintentionally. This is not to say that notions like "cycle" should be foreign to data structures, but rather that they should be explicit; knowledge of such structures may then be utilized more effectively by compilers and interpreters intended for the language.

Data-to-program pointers are not as obviously misused as the other two types, principally because of the extremely limited capabilities generally provided for such pointers. Naturally, switches in Algol inherit all the problems associated with gotos. However, it will be shown below that the notion of "partially instantiated function" actually generalizes this notion of pointer, yet retains the control required; i.e., this is a type of pointer which is not complex enough.

Any programming language with a reference concept is in one sense providing an "assembly language" for data structures, without providing controlled alternatives for expressing sequential relationships. Indeed we lack knowledge of such alternatives. Those languages which do attempt to control pointers usually do so using type structures (modes). Such facilities often prevent useful data structures from being defined (for example, rings) or permit uncontrolled, arbitrarily complex structures to be created.

This should not be misconstrued as an indictment of pointers in general; no one would propose a gotoless language without the ability to define and reference parameterized subroutines by name, which indeed does constitute a use of a program pointer as well as data pointers. It is more the incompleteness of structuring mechanisms for data with respect to pointers which magnifies the problem.

- -

†In a language without pointers, the compiler can determine information about which variables can change over calls. Such a determination is directly analogous to internally reformatting programs with gotos into the gotoless format for optimization purposes.

3. Pointerless data representation

In order to remove the ability to use explicit pointers in programs, we consider the possiblity of using the gotoless constructs directly; i.e., the hierarchical, nested sequential ("embedded lists"), static structure of programs is examined as a base for data structures. Indeed, such structures form the basis of data structures for several higher-level languages. The combination of sequential structures with the gotoless sequencers is indeed richer than the static structures alone. For example, a loop is a gotoless construct for programming languages whose analog in data structures is a cycle--a structure which must be simulated or constructed in languages with the same static structure base.

There are several problems which data structures present beyond those encountered with programs themselves. In particular, a data structure frequently requires that different sequential structures be mapped onto the same entity; the utility of the concept of the "reverse" operator should illustrate this sufficiently. Also, data structures tend to be dynamic; for example, insertion and deletion of elements are operations appropriate to data structures. However, such problems with respect to programs do exist and are beginning to be considered in "incremental compilation" studies within conversational language research [MI].

Note also that the notion of data structure is analogous to program structure in the following sense: although programmers define many different programs they are all considered to be instances of the same program structure--usually described syntactically by some formalism such as BNF. It would appear that to define different data structures requires, in effect, a specification language like BNF in the language. That extensible language advocates often propose this for programming languages suggests that such a mechanism should probably be a shared data structure and program structure extensibility mechanism. It also suggests that a data structure facility founded purely on a single syntax for data structures--as there is a single syntax for a programming language--must be examined critically before proceeding to extension mechanisms. To emphasize, just as gotoless language enthusiasts add new gotoless constructs instead of resorting to the inclusion of the goto to obtain a desirable effect, we prefer the definition of a new construct to provide the effect for which the inclusion of explicit pointers might be proposed.

Languages rarely have a sequential data structuring mechanism; sequence in languages like Algol is induced by the program in terms of an alternative data structure mechanism, the array. Although sequential operators occur in APL, their presentation is as a convenience for describing and restructuring the parallel structures which constitute the structure base. Languages which do offer sequences explicitly as data structure units are the string languages--where concepts of sequentiality and relationships of sequences are constrained to a single level, or else revert to the pointer-chasing mechanisms of other languages, such as LISP. Some of the more modern languages, like Pascal, while approaching adequate inclusion of controlling statements for data structures, continue to represent the relationship between program and data as controller (program) to controlled (data). In what follows, a much more unified view will be established, and, in fact, the distinctions between program and data become pleasantly "fuzzy".

4. Cosequential decomposition: program/data structure correlation

The correlation between data structure and program control structure has been ignored to a large extent in programming language design. An example from APL may help to illustrate the notion involved. In implementing

$A + B + C$

for conforming vectors A, B, and C in APL, a temporary vector, T, may be used in a loop to compute $T_i = B_i + C_i$. The expression's value may be computed in a subsequent loop over $T_i = A_i + T_i$ (an implementation which closely matches the semantic description). However, noticing that the result sequence of the first loop is "cosequential" with the program sequence of the latter loop, the implementer is free to merge the two loops into one, in which $T_i = A_i + B_i + C_i$ is computed. In essence, the loop is used to define a sequencer for the data structures (T, A, B and C) with which the sequence of executions of the loop body is cosequential.

Although the APL programmer should be able to rely on such implementation efficiency [AB], he may have doubts about an expression such as:

transpose (2, (rho A)) rho (A + B), A - B

where loop incarnations of

$(T_{i1} = A_i + B_i; T_{i2} = A_i - B_i)$

would be the desired cosequential program elements. It is not necessary to understand this example. The point is simply that a significantly complex APL expression may have a relatively simple cosequential decomposition not likely to be discovered by a compiler.

Even if a programmer recognizes program/data cosequentiality, in traditional languages he cannot specify the correspondence concisely. For example, many programs produce result (data) sequences with which they are cosequential. The following Algol program is such an example, producing the Fibonacci sequence in T:

```
begin
  T[1] := 0; T[2] := 1;
  for i := 3 step 1 until INFINITY
    do T[i] := T[i-1] + T[i-2];
end
```

Note that there is a program element (statement) execution corresponding to each value in the result sequence. The correspondence is not apparent--the resulting data structure is subordinate to the control structure of the program.

Although the above examples tend to indicate cosequencing of a somewhat trivial form--with program loops--examples dealing with more complex structures are common. Recursive functions implementing top-down and bottom-up tree-scans generally admit identificaton of a recursive program control structure with a nested data structure (a 1-1 sequential mapping between recursion points and tree nodes). Once again, in most languages, the nested data structure is subordinate to the recursive control structure; the sequencing for the data structure is explicit in the program statements themselves.

The extent to which cosequential relations exist in programs and the effect on the programmer's conceptualizaton of them is of primary interest in the language basis developed below. The notions of "partially instantiated function" and "sequence generation" are found useful for emphasizing the cosequential relations mentioned above. Hence, a short discussion of each is necessary before proceeding to particulars of the basis.

The "partially instantiated function" is simply a function with only part of its actual parameter list specified (bound). For example, if "a(i,j)" represents a function which returns the jth element of the ith row of some (implicit) array, "a(,2)" may be used to represent the function "c2" defined by "c2(i) = a(i,2)". This ability to partially instantiate functions--in this case to the second column of the array--has obvious consequences with respect to program generality. For example, if a function "q(x)" expects a vector argument, use of "q(a(,2))" eliminates the need to reprogram q to deal with columns (or rows) of arrays.

Partially instantiated functions are called "sections" in mathematical literature [RO], and we adopt the term here for convenience. The nature of sections is ambiguous: they are both program and data, and attempts to define them as one or the other rely on a preconceived implementation. By themselves, sections do not aid the study of cosequencing; however, sequences of sections will be seen to represent a "middle ground" between data structures and program structures, and constitute a large portion of cosequential result sequences in programs in the basis.

5. Implicit sequence generation

The notion of implicit "sequence generation" is also separate from cosequentiality, but affects its utility tremendously. It arises principally from a desire to express infinite† structures in programming languages as entities which can be dealt with operationally. For example, the Algol version of the Fibonacci sequence generation given above never terminates, and hence does not constitute an algorithm. Conceptually, humans are able to cope with such a sequence; we know that any actual use of the Fibonacci sequence would require its termination. Once again, in traditional languages, program generality is limited, because for each condition of termination of the loop sequence, we must write a separate version of the program which produces it. Alternatives such as passing the termination condition as an argument to the procedure for the Fibonacci sequence, or pulsing a function which always produces the next Fibonacci element, are unacceptable--the conceptually clean notion of an infinite sequence is dirtied by termination mechanisms from within. We are able to write the function we want (as above), but cannot use it!

In the basis presented below, the notion of implicit sequence generation permits the definition and use of infinite sequences. They are terminated from "without"; i.e., boundedness can be a property of the context of the use of a function, not necessarily of the function itself. This ability is useful from a practical as well as the more mathematical standpoint above. Input sequences to programs, operating system state sequences, interrupt sequences, etc., are all realistic sequences which may never be dealt with as sequences in programming languages, other than via explicit pulsing of their generators. It will be seen that the coroutine mechanism required to implement implicit sequence generation represents the beginnings of efficient implementation for (frequently inefficient) algorithm decompositions prescribed by structured programming

- -

†We do not distinguish between (conceptually) "infinite" and (actually) "finitely unbounded". When dealing with mathematical properties of programs, infinite is more appropriate; when dealing with the algorithmic nature of the programs, the latter is appropriate.

studies.

Constraints: a language "basis"

The principal goal of this thesis is to present a programming language basis (the direct analogy to a linear vector space basis is intended). In essence, orthogonal elements of program representation are examined and mechanisms for relating them presented. The dimension of our space is unknown when defining elements of programming. At the very least, technology will add dimensions; at most new language designers can expect only to define a subspace of programming languages. In one sense, the language space is spanned by any computationally complete set of programming constructs. This is not of interest here. Rather, a language construct is independent of a set of constructs according to some intuitive or explicit measure of difficulty of programming without the construct. For example, recursion is a dimension independent of those spanning the FORTRAN syntax.

Continuing the analogy to vector spaces, addition of a new dimension should require reformulation of the existing basis to insure that basis elements remain orthogonal. Orthogonality corresponds (again intuitively) to the optimal introduction of an independent element, which certainly involves maximizing the independence of the construct. This is in turn related to involution, conciseness and consistency. Indeed, many languages are designed with a formal base of primitives--LISP 1.0 [McC], Pascal [WI] and Algol 68 [vW,1969] are certainly such languages. However, once a language is in wide use, in order to avoid reprogramming, independent constructs must be reformulated in terms of the base language, instead of reviewing existing concepts in terms of the addition. New constructs must be "tacked on" as consistently as possible, and orthogonality is rarely achieved.

Partially to avoid this phenomenon, the basis for programming languages proposed below is not viewed as complete. Any reformulation of the language elements to insure consistency with aspects of language design is considered appropriate. Thus, the distinction between a "basis" and an extensible language "kernel" is intentional: modifications to a language based on a kernel arise through extension, not reformulation of the kernel itself. This concept of design methodology eliminates from the outset attempts to incorporate the concepts into existing languages. We reiterate: it is extremely difficult to continuously reformulate new bases of computation to include orthogonal concepts in this manner (i.e. reorienting dimensions to insure orthogonality is difficult).

However, the thesis is very optimistic; for a fixed performance level, an order of magnitude improvement in general purpose higher-level languages for large classes of problems is attainable. And this improvement will be gained in large part by reconsiderations of our computing basis. Arguments are presented which indicate that additions to and reformulations of the basis presented should accomplish this end.

Approach: a Sequential, Applicative Language Basis

In the discussion about sequences above, it was noted that the consequences of the inclusion of pointers are particularly unmanagable when combined with "assignment" to produce side-efects. Formal applicative languages (LISP [McC], lambda-calculus [CH]) do not suffer from this defect. The basis is therefore set in an applicative language framework. The store operator has been introduced in applicative languages (e.g., as in Gedanken [RE,1970]) with the result that favorable properties of the applicative language are lost. In a sense, side-effects ultimately appear in our langauage base in a controlled manner--the assignment statement does not.

The applicative language chosen is not conventional, but rather based on operators instead of functions. The expression is the fundamental unit of a program (it may evaluate to a sequence), and consists of a sequence of operators and operands, with left-to-right precedence in evaluation. The choice of operators over functions is significant from a syntactic and notational viewpoint, though both provide the temporary name minimization aspects requisite to notational conciseness. The language is typeless: types are implicit with the input format. A primitive operator definition facility is introduced, with scope rules unspecified. This has the effect of avoiding scope "tricks," or rather postponing the decision of which tricks to prefer. Formal considerations are not of interest here, so arithmetic, relational, and boolean operators are considered primitive.

1. **Homogenous sequences**

We eliminate the ambiguities intrinsic in allowing the notation for a sequence to be the same for programs and data--the former often prefer the value of their last expression, instead of the entire computation sequence of the program as value. Aspects of data which differ from program sequences--including creation by algorithm, and insertion and deletion of elements are then considered. There is considerable flexibility in the basis here; the constructs are initially merely chosen to be consistent with the operators described below. Alternative and possibly preferable formulations can be made in this area.

The gotoless constructs are then introduced as operators over any type of sequence--program, data or a combination. These consist of selection, conditional, loop and escape facilities. Careful examination has shown that two operations for combining data and program sequences, in conjunction with the above language constructs, provide a powerful basis for computations over homogenous sequences. (In a sense, only applicative languages should be compared to this basis; however, real programming languages are compared in order to determine the directions of further extensions to the base as well as to examine the form of conventional structures and constructs in terms of the new base.)

The operators accomplishing this power are actually quite simple, and aspects of them occur frequently in existing languages (which is why they were chosen). The coapply operator--"."--simply combines two sequences, one element at a time, evaluating to a new sequence representing the combination. For example, if we have a sequence of unary operators

$<u_1; u_2; u_3>$

and a sequence of arguments

$<a_1; a_2; a_3>$

then

$<a_1; a_2; a_3> . <u_1; u_2; u_3>$

is the sequence

$<a_1\ u_1; a_2\ u_2; a_3\ u_3>$,

where "$a_i\ u_i$" is evaluated before "$a_{i+1}\ u_{i+1}$". When the operator sequence is a repetition--$<u_1; u_1; u_1>$--the more conventional concept of distributed operator should be recognized.

The second operator actually embodies the notion of sequence. It is most easily derived by examining the consequences of removing the assignment statement from a traditional language such as Algol. Clearly, one realistic interpretation of a program is then the sequence of values of each expression. The more traditional meaning is the value of the last executed expression. In a language without the assignment statement or escape facilities (return statement) this is always simply the last expression (preceding expressions need not be evaluated, for they can have no effect on any other expression in the same sequence). Inclusion of a return or escape operator merely requires that the conditional of each such operator be examined sequentially; the consequent of the first to escape is the only expression that requires evaluation.

Hence, a mechanism which establishes a relationship between a given element and its predecessor must be provided. The accumulation operator--"/"-- accomplishes this by producing a sequence, each element of which is a function of its predecessor. An initial value must be given along with the sequence of functions to be applied. For example, if "s" is the (left-unary) successor operator,

```
0/<s;s;s>
  ≡ < 0 s; (0 s) s; ((0 s) s) s> is
  ≡ <1; 2  3>.
```

This operator is particularly related to the APL reduction operator--whose symbol it shares--and to the notion of regular automaton. Its name derives from the action of machine instructions on an accumulator, which it simulates.

2. Non-homogeneous sequences: the "recursionless" constructs

The above considerations yield a langage basis which is quite concise for homogeneous sequential structures. However, algorithms dealing with non-homogeneous structures are not nearly as succinctly expressible. Traditionally such structures are best dealt with using recursion--either via recursive function calls or a recursive data structuring facility, or both. The awkwardness of recursive expressions using the sequential constructs leads us to the (obvious) conclusion that recursion is indeed orthogonal to strict sequentiality and fundamental to the facile treatment of indefinitely nested sequences.

However, a recursive function definition facility never enters the basis (nor does any equivalent, such as the LISP "label" facility). Instead "corecursive operators", analogous to cosequential operators ("." and "/" above) for recursive structures are introduced. Program and data structures in which a one-to-one identification can be made with recursion in the data structure and recursion in the program abound in programming. For example, the correspondence between tree nodes and recursive functions is quite obvious in "top-down" and "bottom-up" algorithms.

A LISP example will help to illustrate the concept†. The function "D" is defined below for a list "L". The value of the function is simply a list similar to "L" with the function "d" applied to each of its elements:

```
D[L] = [null[L] ⇒ NIL;
        T ⇒ cons[ d[car[L]]; D[cdr[L]]]].
```

The non-trivial consequent in "D" is an expression with precisely the same structure as "L", but with "d" applied to each element. In particular, each time "D" recurs, "L" nests (its "cdr" is taken). Thus, the recursive structure "L" is corecursive with the recursive structure of the function "D".

The LISP example illustrates an essentially sequential algorithm which should not even be dealt with recursively. We develop operators which are able to deal with considerably more complex recursive structures. In particular, recursive analogs to the "." and "/" operators are developed which deal with quite general recursive evaluation structures. We are lead to an analogy between the "gotoless" constructs and "recursionless" constructs: viz., recursion is implicit in the corecursive operators. Their generality leads us to consider removing explict recursion from programming languages and defining "recursionless" languages.

3. **Cosequential decomposition and "coroutineless constructs"**

Even with the "cosequential" and "corecursive" operators, the basis is unable to express some algorithms well. We should expect algorithms whose conceptualization hinges on issues orthogonal to nested-sequences to present difficulties. These might, for example, be algorithms in which parallel structures or random access mechanisms are required. However, there are some algorithms which are clearly in the domain of "nested-sequential" algorithms, but which simply cannot be expressed well.

Partially as an effort to study such algorithms and partially to indicate how the language basis can be implemented, we relax the "cosequential" assumptions about programs and data and introduce "partial cosequentiality". This ultimately leads to the introduction of "coroutines" into the basis. The initial cosequential operators are found to be easily implementable in terms of this more primitive control/selection mechanism. Ultimately, we recognize the true nature of the "cosequential" and "corecursive" operations as "coroutineless" constructs. Hence, in much the same manner as with the goto and recursion, the coroutine facilities ultimately defined are presented only as a low level mechanism to be used to define a richer set of "coroutineless" constructs. That is, we advocate the removal of the explicit coroutine call at a future date, but

- -

†This example is too trivial to illustrate the actual corecursive operations of the basis; this particular problem should never even be considered in a recursive context. In fact, "maplist" in LISP can be used to accomplish this effect; i.e., this trivial form of corecursion has been recognized in LISP.

present it as a tool for studying extensions to the set of "coroutineless constructs".

The remainder of this work deals with the material motivated in these last three sections in considerably greater detail. In Chapter II the initial portion of the basis dealing with strictly sequential effects is laid out. Recursive considerations--"recursionlessness" and "corecursive operators" constitute Chapter III. The coroutine as the fundamental mechanism involved in the implementation of the basis is the subject of Chapter IV. In Chapter V, we draw conclusions about the significance of the reformulation of programming that the basis entails, and indicate future directions for its development.

CHAPTER II

THE INITIAL BASIS

Programming language design decisions are often involuted and interdependent. Frequently a distributed set of decisions must be made in order to satisfy a single language design criterion. This is particularly characteristic of a new language basis, where each conventional language construct must be reexamined or reformulated.

As was indicated in Chapter I, the germinal decision of this work is to control pointers in higher-level languages by applying the gotoless constructs to "nested-sequential" structures in general. At the outset only gains in expressiveness were envisioned, gains analogous to those provided by the gotoless constructs in traditional languages. In such languages, expressiveness benefits accrue from the provision of a hierarchical decomposition to programs and the enlargement of the common vocabulary for description of control beyond the primitive state represented by the goto. In removing the pointer from languages, the only indication of potential gains in efficient implementation--a language design criterion--is that the gotoless constructs in languages do provide efficiency gains in the context of program control structures; this must be weighed against the compelling reason for the inclusion of reference variables in higher-level languages--namely, efficiency!

All decisions made in the design of the basis contribute toward an enhancement of expression--concise representation. The decisions may be categorized as either fundamental or syntactic. Fundamental decisions are of particular interest and will be dealt with most thoroughly, particularly with regard to the design criteria of expressiveness and efficiency. The studies of structured programming and program optimization provide concrete methodologies for determining whether a decision satisfies these vague criteria. The nature of the language "basis" is such that fundamental decisions should not be altered when designing a language from the basis; syntactic decisions may be.

Although syntactic considerations are not considered paramount for development of this basis, the principle of "involution" is adhered to, and an some sense, exaggerated by what we term "combinatorics". "Involution" refers to the internal consistency of language primitives with respect to the ways in which they combine and admit substitution of other primitives. Expression languages illustrate the principle nicely, in the sense that any expression in a program may be substituted syntactically for any other expression, with only problems of variables' scopes to be considered. Hence, if computation A depends on the result of a computation B, the particular form of computation B need not be known in an expression language to implement A. In a statement language, A will depend on whether the result of B is from a loop, and, hence, left in some temporary about which A must know, or whether it is from a function or expression, and can be computed directly.

By combinatorics we mean the way in which primitives interact to form the "most natural" result. Only recently have languages which use combinatoric notions extensively come to popular consideration by the language design group at large, with Backus' "Reduction Languages" [BA,1972], [CU] and the resurrection of Aiken's "Dynamic Algebra" [NO]. However, combinatoric aspects of almost any language can be discerned, and a few examples may aid the reader in understanding this notion (which is admittedly vague conceptually, though not formally).

The LISP interpretation of non-NIL as the true condition in COND represents a combinatoric decision. In SNOBOL, the ease with which a variable is assigned to the portion of a pattern matched by an arbitrary string within the pattern may be construed as a decision which aids SNOBOL's combinatoric power. In Algol 60, the ability to have an **if-then** statement is of similar utility. In particular, in LISP, the designers predicted:

```
(COND ((NONEMPTY X) E1) ((NONEMPTY Y) E2) ...)
```

(where NONEMPTY has the more traditional, strict T/NIL value) would be the most useful form for the condition, and chose to allow:

```
(COND (X E1) (Y E2)...).
```

In SNOBOL, syntax for the pattern-element by pattern-element detachment of substrings to obtain the matching arbitrary string is clearly less concise than that actually chosen to accomplish the effect. And obviously, the repeated use of "**else** dummy := 0" in

Algol--the alternative were there no **if-then** construct--is less concise and represents a probable frequent use of the conditional.

To generalize, combinatoric decisions involve the elimination of some syntactic constructs from a preconceived model, to favor the most frequent use of that syntax. Hence, it is particularly related to the notion of "defaults" and tends to enhance involution. In general, combinatoric effects must be considered in the design of primitive constructs. The nature of combinatoric and involutionary decisions is such that they may be described best after all constructs are known. Hence, in what follows, frequent reference is made to "combinatoric reasons" for the particular format of a primitive construct. The nature of these decisions will be presented after the entire basis has been elaborated.

The basis described below is an "initial" basis, a "final" basis will be developed in successive chapters. The presentation is not formal, but rather emphasizes the nature of the decisions which produced the initial basis. As such, the description should be viewed more as initial considerations toward a language design, rather than as a language specification.

Fundamental Decisions in the Design of the Language Basis

The design of the basis is presented below under major headings which reflect the fundamental decisions involved. First, the decision to use an applicative language is examined. In order to establish a universe of discourse, the primitives are presented next, although the fundamental decisions involved here are combinatoric in nature. The gotoless constructs are then considered and introduced into the basis.

The cosequencing operators introduced in Chapter I are then considered, followed by a discussion of the notion of sequence "generation". After rehashing the cosequencing operators in light of sequence generation, the combinatoric decisions involved in all of the major decisions are considered briefly.

1. An applicative language

An applicative language framework is chosen principally for its tight control over pointers and related concepts. This choice is a fundamental language design decision. The equivalence property which makes such a language desirable from both a structured programming point of view and an implementation (optimization) viewpoint is simply: identical expressions in the same static context have identical values. From an optimization viewpoint, this permits multiply recurring common subexpressions to be evaluated once, and for all but one instance to be replaced by a direct reference to the contents of a cell containing the value.

Structured programming is concerned with the control of relations which do not remain invariant over an expression. In particular, if we characterize the execution of a program by its effect on its environment--a dynamic description of the name/value associations available to the program [RE,1972]--the effect of evaluating an expression in applicative languages is simply to extend the environment of the caller. Hence, any relations which fail to hold in the new environment are due solely to the "addition" of the new value to the old environment. The impact of applicative languages to structured programming lies in the localization of the dynamic effects to the environment of evaluating a function application.

Naturally, applicative languages are no panacea to these two studies. It is often extremely difficult to optimize functions in applicative languages to a point comparable with corresponding algorithms for sequential languages. This difficulty arises in part from the complexity of "untangling" the control/data space when dealing with recursive structures of some complexity (see Chapter III). Additionally, even though an iterative algorithm may be derived from a recursive specification, a creative "leap" to a more efficient algorithm may be masked by the recursive structure, even though it is quite clear from the sequential specification (see example at the end of this chapter). Structured programming benefits from sequential languages to the extent that invariant relations may be found over environments which do change drastically. The inclusion of sequential constructs in the basis will permit use of positive aspects of both sequential and applicative languages.

An operator version of an applicative language is chosen principally for notational convenience arising from both combinatoric and operand-evaluation-sequence considerations, which are best described after definition of the basis. The operator notation, which allows nullary, left-unary, right-unary and binary operators, is also more general than a functional notation. In particular, given the ability to pass sequences as operands, left- and right-unary operators may be extended to pre- and post-fix functions, respectively, simply by providing a matching mechanism for the parameter sequences at the function definition site. Also, although a left-to-right expression parse/evaluation order is chosen, a more elaborate precedence structure may be applied when operator relationships are better understood. Hence, the operator notation is chosen initially simply because of the flexibility it admits.

The metalinguistic operator definition notation used is as follows†:

```
Left-Formal  Op-Name  Right-Formal :: Defining-Expression;
Left-Formal  Op-Name               :: Defining-Expression;
   none      Op-Name  Right-Formal :: Defining-Expression;        [1]
             Op-Name               :: Defining-Expression;
```

corresponding to binary, right-unary, left-unary and nullary operators, respectively. For example, a right-unary identity function may be expressed:

```
x id :: x                                                          [2]
```

A binary identity function which ignores its left-operand and returns its right, is:

```
x rid y :: y                                                       [3]
```

The "reserved word"†† or "token" **none** resolves the ambiguity of two consecutive names in unary operator definitions.

- - - - - - - - - - - - - - - - - - - -

†Operator and formal names are any combination of decimal digits and upper or lower case alphabetic characters; sequences of special characters are also permitted as operator names (see Chapter III).

Use of an operator name in its defining expression does not constitute a recursive call, but rather refers to the previously defined operator of the same name (which this overrides). This standard extensible language interpretation is chosen to permit redefinition of functions, a useful device in the presentation below. Operator definition is initially "metalinguistic" to avoid scope considerations; an operator definition facility must enter the basis at a later time.

A left-to-right expression parse is chosen over a more elaborate precedence scheme as a concession to inexperience with the unusual operators of the basis. Although, in fact, a reasonable precedence could be proposed, its presence would cloud the presentation of the basis. The choice of a left-to-right scheme over APL's right-to-left scheme is primarily to remain consistent with the accumulate operator (see below); the nature of "generation" precludes its right-to-left evaluation.

2. Primitives

We are not overly concerned with the particular primitive operand and operator types of the language. However, we assume a "typeless" language (like APL) for the generality it provides; we may define operators which apply to different types of operands, depending only on primitive relations defined on the types. Hence, it is probably more accurate to state that the types of a language developed from this basis should involve sets of relations or functions defined over what are more traditionally thought of as types. For example, in a "typeless" language, the function

min(a,b) = if a < b **then** a **else** b

has meaning only if the relation "<" has a boolean interpretation for the pair (a,b). This is the case for any combination of real and integer variables in Algol. A typed language

- -

††In a language developed from this basis, reserved words should be present in the initial symbol table, and the ability to override their definition provided. This is the preferred extensible language interpretation [RE,1971].

would insist that four separate functions be defined to cover all cases of this expression. Subsequent definitions in terms of "min"--e.g.,

$$g(a,b) = a \div \min(a,b)$$

--also require four definitions; the loss of generality can be exponential in the number of distinct types. Hence, the basis is typeless. (The combinatoric implications of typelessness are addressed below.)

The operand types initially present in the basis are

	Type	Examples	Generic-Variables†
1.	Possibly Negative Integer (PNI)	1, 36, ~125	i,j,k,m,n
2.	Character	"a", "B", """"	c,d
3.	Sequence	$<el_1; el_2; \ldots; el_n>$	s,p
4.	String	"b+""" ≡<"b"; "+"; """">	u,v
5.	The Empty Element	nil.	

We leave the description of sequence elements unspecified here, but, of course, allow any instance of an operand type. In particular, the ability to nest sequences is primitive. (The generic variable "t" will be used when dealing specifically with nested-sequences--see Chapter III.) A distinct unary minus, "~", is adopted here for a reason which will become apparent under the discussion of "section" below.

- - - - - - - - - - - - - - - - - - - -

†These names will be used in examples throughout the text.

The arithmetic operators--+,-,mul,div, and mod--are primitive operators with results of type PNI defined only on operands of type PNI†.

The relationals--le,ge,lt,gt,=, and ne--are defined over pairs of PNIs and over pairs of characters (the colating sequence is presently left undefined)†. The definition of relationals, however, is not traditional. If "rel" is a relational, then "a rel b" has the value "a" if the relation holds, nil otherwise. The important decision here is combinatoric in nature; it is important that one of the operands be chosen as the value of a true relation.

A unique primitive operation which produces primitive operators is also permitted; this operation is termed "partial instantiation". The "section" or "partially instantiated function" was motivated in Chapter I as a natural mechanism for expressing data structure concepts of restriction. In fact, they play a much more significant role in the basis in that many programs are sequences of partially instantiated functions. In particular, we allow the partial instantiation of any binary operator to produce either a left- or right-unary operator. For example,

```
x (-3) ≡ x - 3;
(4 mul) y ≡ 4 mul y;
uminus :: 0-;
uminus 3 ≡ (0-) 3 ≡ (0-3) ≡ ~3.
```

As an aid to involution, we extend partial instantiation to include any expression missing an operand on the left or right:

```
x (-5 mul 3) ≡ (x-5) mul 3
(4+(3 mul)) y ≡ 4 + (3 mul y)
```

The parenthesization of the operator expression is preserved, as indicated in the second example. The ability to instantiate is uniformly allowed with any binary operator, including those defined in the metalanguage. In general, operators are

---

†It is convenient to have the relationals and arithmetic operators defined also on nil, but motivation for the particular choice must be delayed until after the "conditional operators" have been set forth.

permitted as sequence elements. (A more precise formulation of the rules of composition and instantiation appears in Appendix I.)

3. Gotoless constructs

Application of the gotoless constructs to nested-sequential structures represents the second "fundamental decision" toward the design of the basis. The particular constructs of interest here are: subroutine call, grouping, selection, conditionals, looping facilities and escape mechanisms. As has been stated previously the single property preserved by the gotoless constructs is program hierarchical decomposition. The impact of this property to structured programming and optimization studies arises from the ability to identify non-primitive program elements which have a single predecessor.

From a structured programming viewpoint, properties (relations on the environment) preserved or destroyed over program element execution are of interest. Gotoless constructs allow element identification to include larger elements than single program statements. The gotoless constructs present a single predecessor to each program element other than subroutines and loop bodies. The relations considered for any given program element are localized to consequents of those holding after the predecessor's execution. The predecessor may be hierarchical, as in the case of selector to selection and boolean to conditional, or sequential as in the case of grouped elements. Loops are given hierarchical dependency on the negation of relations implied by the termination condition. Hence, with the exception of the subroutine call, consideration of predecessor relations is a linear process in a gotoless language, whereas, inclusion of the goto potentially requires exponential considerations.

Optimization considerations often center about equivalence relations preserved on environments; hence, any localization of the considerations of these relations aids efficient implementation. Any structured programming efforts which localize considerations of arbitrary relations on the environment will usually localize considerations of equivalence relations as well, and hence aid efficient implementation! This is borne out in Geschke's thesis on omptimization [GE].

The gotoless constructs enter the basis as operators. This is simply for consistency with the rest of the basis and has extreme combinatoric significance. It will be seen that some of the operators are more of the nature of applicative language "forms", in that they allow operators as operands. However, consideration of them as operators is useful presently.

The first gotoless construct to be considered--selection--arises in part from an ambiguity introduced in allowing both programs and data as sequences. In allowing programs as sequences and sequential values, we have a notational choice: we can distinguish between program and data sequences by distinct bracketing pairs, or not.

We choose not to make the distinction; i.e., type sequence above may contain program (operator) or data elements. This allows the flexibility of operating on programs as we do on data, without an additional conversion mechanism. However, this decision immediately presents an ambiguous interpretation for a program sequence in light of the recent sequential language interpretation of its value as the last program element executed [RE,1970], [WU,1971]. For example, if

begin e_1; e_2; ... ; e_n **end**

is a compound in an Algol-like expression language, where the "e_i" are expressions and none escapes, we may choose to interpret it as representing

$<e_1; e_2; ...; e_n>$

or

e_n

(the traditional interpretation). The sequential interpretation is chosen. If the last value is desired, an explicit operator--**val**--must be applied.

$<e_1; e_2; ...; e_n>$ **val** $\equiv e_n$.

In a sense, the **val** operator is the only form of selection in the basis, although only the last element of a sequence is ever selected by it. Although the select operator,

s **sub** i ≡ s_i (the ith element of s) [4]
when i is in the range [1, length of s]

≡ nil otherwise,

will be used frequently, it will later be demonstrated that the operator is derivable from the initial basis.

The ability to derive operators from the basis is considered important in the context of extensibility; i.e., although inclusion of a large number of primitive operations in a language is desirable (see Chapter 1), a layered description of such operations is considered invaluable to understanding the language. Additionally, criteria for entry to the basis of extensible operators should be developed; in particular, **sub** would enter the basis for implementation reasons, to be described later.

A second ambiguity arises when we consider interpretation of nested sequences in higher-level-languages. Again, if

begin e_1; **begin** e_2; e_3 **end**; e_4 **end**

is a compound, we can choose either of:

<e_1; e_2; e_3; e_4>

(the actual program interpretation sequence) or

<e_1; <e_2; e_3>; e_4>.

The latter is chosen to include the gotoless construct for grouping as the already present sequence brackets. To obtain the former interpretation, the **gen** operator must be applied to the subsequence whose elements form a continuation of the supersequence:

<e_1; <e_2; e_3> **gen**; e_4> ≡ <e_1; e_2; e_3; e_4>.

For example, we may write a concatenate function:

s conc p :: <s **gen**; p **gen**> [5]
 ≡ <s_1; s_2; ...; s_n; p_1; p_2; ..; p_m>

Notice that **gen** only applies to the sequence which is its argument, and not to its argument's subsequences; i.e.,

<1> conc <2; <3;4>; 5> ≡ <<1> **gen**; <2; <3;4>; 5> **gen**>
 ≡ <1; 2; <3;4>; 5>.

Note also, if s is a sequence,

<s **gen**> ≡ s, and hence, <<> **gen**> ≡ <>.

A third group of gotoless constructs, the conditionals, may be thought of as "sequential booleans". In order to express the conventional if-then-else control construct operationally, **then** and **else** operators are derived. Again for combinatoric reasons, the LISP boolean is used; in any context where a boolean occurs, the criterion for validity is that the result be a primitive other than **nil**. This is consistent with the relational operators described above. Additionally, any operation which would traditionally produce a boolean result must produce a non-empty element or **nil**. The choice of this non-empty value distinguishes the conditional operator definitions:

x **else** y ≡ y when x is empty,
 ≡ x otherwise;

x **then** y ≡ y when x is non-empty,
 ≡ **nil** otherwise.

Unlike LISP, no value "T" for true is part of the language. If we consider boolean functions over the set [true, **nil**], where we define "true" as any non-empty value (e.g. 1) the following definitions are apparent:

```
and :: then
or  :: else
```

However, there is no way to construct the "not" connective in this context. This is indicative of a failure in completeness of our gotoless constructs: there is no combination of **else** and **then** which evaluates to "y" precisely when "x" is empty in "x conditional y". We could include a "not" function in the basis as having some random non-empty value. However, the basis is biased towards use of non-empty results of conditionals as expression values, and we would most frequently use "not x **then**". Thus, the **excludes** operator is defined to permit this effect:

```
x excludes y ≡ y when x is empty,
             ≡ nil, otherwise.
```

With this we can complete our boolean repertoire, by defining:

none† not x :: x **excludes** true.

(Only the conditionals **else** and **excludes** are necessary. For a more complete description of the somewhat strange ramifications of this logic system, see Appendix II.)

A fourth gotoless construct is the loop; unlike other languages, a set of terminating facilities are not presented implicit with the looping construct. The loop operator, "*" (the "Kleene star"), continuously replicates its argument until terminated implicitly, or by using the escape operators:

```
x * ≡ <x; x; x; ...>
```

In one sense, this operator is taken from data structure specification, where use of a pointer to implement a cycle is common. Naturally, the interpreting program normally must impose the interpretation as a cycle, and must terminate such an interpretation explicitly. For example, a function which produces alternately its left and right argument is:

- -

†See [1] above.

```
x alternate  :: <x; y> gen *;                          [6]
             ≡ <<x;y> gen; <x;y> gen; ...>
             ≡ <x; y; x; y; ...>.
```

The choice not to require explicit termination mechanisms was made in the hope that frequently the termination condition can be implicit to the usage context of the non-terminating sequence. As discussed in Chapter I (and also below), termination of generated sequences is factored as not intrinsic to loops **per se**, but rather to generated sequences independent of the generating mechanism. For example, we might choose to define operations on a rational number representation formed by:

```
u rat v :: u conc† (v gen *)
        ≡ <u gen; v gen * gen>
        ≡ <u gen; <v gen; v gen; ...> gen>
        ≡ <u_1; u_2; ...; u_n; v_1; ...; v_m; v_1; ...; v_m; ...>

e.g., "3.7" rat "23" ≡ "3.7232323..."
```

For some operations, the decision to terminate sequences thus formed will not be based on properties of the repeated digits, but on precision considerations; i.e., the termination is not a property of the loop, but rather of the context of the sequence generation. It is worthy of note that the same functions may then be used on sequences generated by irrational number generators; i.e., the complications of termination are localized to the use of the generated sequences, not to the various generation mechanisms for the sequential arguments.

Explicit termination of sequences is accomplished with the use of the escape (gotoless) operators, **exs** and **txs**. The former--**exs**, "else exit sequence"--exits the innermost sequence in which it is embedded when its operand is empty; otherwise, its value is its argument. When the decision to exit is made, the empty element does not contribute to the resulting sequence; e.g.,

- - - - - - - - - - - - - - - - - - - -

†See [5] and also Appendix III for useful functions defined in the text.

<1; 2=2 **exs**; 2=3 **exs**; 4> ≡ <1; 2 **exs**; **nil exs**; 4>
≡ <1; 2>.

This intuitively corresponds to the "while-do" control construct of several higher level languages.

"Then exit sequence", txs, exits the innermost sequence in which it is embedded when its operand is non-empty; its value is nil when its argument is nil. Unlike **exs**, the argument causing the sequence to be exited with txs is included in the result; e.g.,

<1; 2=3 txs; 2=2 txs; 4> ≡ <1; nil txs; 2 txs; 4>
≡ <1; nil; 2>.

The intuitive correspondence with other languages here is with "do-untii", a traditional search mechanism.

N.B. Although generation and factored termination represent a fundamental design decision, the explicit gains are best presented following the discussion of the cosequencing operators.

4. Cosequencing operators

To this point the language basis is not "computationally complete". We have no recursive function definition capability, and although the loop operator, "*", is present, there is no way to relate successive elements of a loop. Before we introduce the necessary operators, notice that the basis includes:

1. An operator language with a left-to-right expression parse;
2. A metalinguistic operator definition notation;
3. Primitive types: PNI, character, sequence, string and empty element;
4. Primitive arithmetic and relational operators;
5. The **gen** operator (form);
6. The gotoless operators: conditionals (**then**, **else** and **excludes**), selection (**sub** and val), loop ("*"), and escapes (**exs** and txs).

Formalists will note an ambiguity in the interpretation of applications of "*" and **gen**. For example, although the form "+(mul 3)" is disallowed as an instance of operator composition, the similar form "+ *" is permitted (each is a binary operator followed by a right-unary operator). One can either envision a set of quote rules, permitting the application of some operators to other operators, an elaborate precedence formulation, or a notion of "form" in the lambda-calculus sense, to resolve the conflicts (see also Appendix I). Presently we rely on intuition guided by the derivations presented.

In Chapter I the coapply and accumulate operators were presented as means for relating programs with their arguments and results. In particular, both operators are forms of "apply" functions--or more accurately, "application generators"--in that they relate a program sequence positionally with a result sequence. The coapply operator, ".", additionally relates its argument sequence with its function sequence:

$$s \;.\; q \equiv \langle s_1\ q_1;\ s_2\ q_2;\ \ldots\rangle.$$

Generation of this sequence terminates with the shorter sequence, iff one argument sequence terminates.

In combination with a loop the conventional distributed operator concept is realized, for example:

$$\begin{aligned} s \;.\; (+1\ *) &\equiv s \;.\; \langle +1;\ +1;\ \ldots\rangle \\ &\equiv \langle s_1{+}1;\ s_2{+}1;\ \ldots\rangle. \end{aligned}$$

More exotic sequences can be expressed easily:

$$\begin{aligned} s \;.\; (+\ \text{alternate}^{\dagger}\ -) &\equiv s \;.\; (\langle +;\ \text{->}\ \textbf{gen}\ *) \\ &\equiv s \;.\; \langle +;\ -;\ +;\ -;\ \ldots\rangle \\ &\equiv \langle s_1{+};\ s_2{-};\ s_3{+};\ s_4{-};\ \ldots\rangle. \end{aligned}$$

- -

†See [6] for definition.

(Appendix III contains a list of all functions whose previous definition in the text are reused further along in the development of the basis.) The alternate "program" may then be used to express more complex data sequences, for example:

s . (+ alternate -) . (1 *)
$\equiv$ <s_1+1; s_2-1; s_3+1; s_4-1; ...>

This instance of section sequence construction illustrates a frequent use of the coapply operator: to create function sequences with potentially complex components; i.e., programs.

The coapply operator is also used for controlling sequence generation. For example, a function which produces a "header sequence" of q--a sequence consisting of initial elements of q in the same order--the same length as another sequence s is:

s controls q :: s . (rid† *) . q; [7]
$\equiv$ <s_1 rid; s_2 rid; ... ; s_n rid> . q;
$\equiv$ <s_1 rid q_1; s_2 rid q_2; ... ; s_n rid q_n>;
$\equiv$ <q_1; q_2; ... ; q_n>.

(when n = length s le (length q)).

Note that the notion of sequence involved here is somewhat trivial; the only truly sequential relationship (as opposed to positionally parallel relationship) of the function (program) sequencer with its data sequence concerns termination. For example, consider the operator:

s rplus p :: s . (+ *) . p;
$\equiv$ <s_1 + p_1; s_2 + p_2; ...>

"rplus" approximates the APL addition operator on two row vectors. However, there is a subtle difference between this operator and the corresponding APL operator. The implicit finiteness of sequences p and q in APL permits "pregeneration" of p and q, and

- -

†See [3] for definition.

computation in parallel of pairwise sums, given a sufficient number of adders on the interpreting machine. The basis does not presume finiteness, and an implementation cannot (in general) pregenerate p and q; a sufficient number of adders cannot be guaranteed to exist. Thus, if we consider it a property of parallel operations that they terminate, the basis does not admit a parallel implementation.

More crucial to the non-parallel implementation of the coapply operator are the escape facilities. Consider the operator:

$$s \text{ nonempty} :: s \;.\; (\textbf{exs} *);$$
$$\equiv \langle s_1 \textbf{ exs}; s_2 \textbf{ exs}; \ldots\rangle \equiv \langle s_1; s_2; \ldots; s_n\rangle$$

where "s_i" are nonempty in the range [1,n], and n is the length of s or s_{n+1} is empty. Here, the parallel pairwise application of the sequence elements is also inadmissable; one cannot determine the operand lengths required until after the result is produced.

Despite the necessarily sequential nature of the coapply operator, the nature of the sequentiality is trivial. No result element depends on its predecessor's value. Traditional sequential languages are sequential for the same reason: statement si modifies the environment in some way on which statement s_{i+1} may functionally depend.

The accumulate operator, "/", introduces this dependency and represents a constrained form of assignment. It is defined for an arbitrary initial value, y, and a sequence, s:

$$y \;/\; s \equiv \langle y\ s_1; (y\ s_1)\ s_2; ((y\ s_1)\ s_2)\ s_3; \ldots\rangle$$

Generation of the sequence terminates when and only when s terminates.

One interpretation of this operator is as the execution sequence of a program on a single-register machine (which has no store operation). The interpretation is not quite appropriate for the register may contain an arbitrary sequence, if desired. The program sequence can, however, have a complex control structure. For example, the Algol program:

```
begin
   real a; integer i;
   a:= 3; i := 0;
   for i := i + 1 while a < 9
     do a := a * b[i];
   if a < 13 then a := sin(z) else a := cos(z);
end
```

can be represented as:

```
3/ < mul * .  b .  (lt 9 exs *) gen;
        (lt 13 then sin else cos) z>

   ≡ 3 / <<mul b1 lt 9 exs; mul b2 lt 9 exs;...> gen;
                 (lt 13 then sin else cos) z>
   ≡ <3 mul b1; ...  ; 3 mul b1 mul ...  mul bn;
         3 mul b1 mul ...  mul bn (lt 13 then sin else cos) z>
```

where accumulated product is less than 9. (The example is only intended to indicate similar complex control structures; however, the final value of the Algol variable "a" will be the same as the val of the latter expression above.)

However, this neither reflects the nature of most programs written in the basis, nor of those written in Algol. Although the successive values of the accumulator may be looked upon as the access sequence for an assignable variable, this sequence is traditionally distributed throughout programs. Here it is not, and, indeed, a reorientation of programming style must occur. The access sequence and the controlling sequence are now equally important.

This completes the initial basis and we are now in a position to exhibit more realistic functions. For example, the positive integers, P, may be represented:

```
P :: 0/(+1*);                                                    [8]
     ≡ 0/ <+1; +1; ...>;
     ≡ <0+1; 0+1+1; 0+1+1+1; ...>;
     ≡ <1, 2; 3; ...>.
```

This is the first example of a useful non-terminating function. In particular, it is clearly beneficial to be able to express infinite sequences in programming languages. Many mathematical formulations rely on such expressions: rational and irrational numbers, infinite series, etc. Obviously, from a programming standpoint, such a function can never be "executed" (control may not transfer to it expecting a return); in fact, it can never be used in its entirety. It is the ability to express non-terminating sequences which leads to the "generative" implementation discussed below. In effect, each sequence expression is considered to be a generator, an instance of which may be "pulsed" to produce elements when needed.

It is the accumulate operator which gives utility to the escape operators. For example, the function:

```
    s while f :: s . (f exs *)                                  [9]
```

is trivial if s is not produced using an accumulation (or as input to the program), for no non-trivial relationship of the elements of s may be established without it. In combination with the accumulation of positive integers, we may produce the first n positive integers:

```
    n pos :: P† while ( le n);                                  [10]
          ≡ < 1 le n exs; 2 le n exs; ... ; n le n exs;
                          (n+1) le n exs; ...>;
          ≡ <1 exs; 2 exs; ... ; n exs; nil exs; ...>;
          ≡ <1; 2; ... ; n>.
```

It is now an easy matter to illustrate that sub need not be a primitive operator; the "head" function is defined which produces the sequence consisting of the first n elements of the sequence s:

```
    n head s :: n gt 0 then (n pos controls†† s) else <>        [11]
```

- -

†See [8] for definition.

The **sub** function simply selects the last element of this sequence:

```
s sub i :: i head s val.
```

which will be **nil** when "i" exceeds the length of "s", and otherwise, the "ith" element.

In fact, this is not an accurate representation of the **sub** function. The "controls" function as defined above presumes that the length of the controlled sequence is at least as long as the controlling sequence. Boundary conditions generally introduce complexity in programming language definitions, especially when extending languages. (User-defined functions need not be "general" when the particular case of interest is known to have certain properties.)

In order to define a new "controls" function which behaves suitably for the **sub** function, the initial functions defined on the empty sequence are considered:

```
<> val ≡ nil;
<> .  s ≡ s .  <> ≡ <>;
y / <> ≡ <>;
<<> gen> ≡ <>.
```

Otherwise, the empty sequence acts as any other sequence; e.g.,

```
<> then 1 ≡ 1.
```

The case in point--"controls"--has an erroneous value when its boundary conditions are not met. For example,

```
<1; 2> controls <4>
   ≡ <1; 2> .  (rid *) .  <4>
```

††See [7] for definition.

```
≡ <1 rid; 2 rid> .  <4>
≡ <1 rid 4>
≡ <4>.
```

Although this may be a reasonable interpretation for the value of "controls", alternative interpretations are equally reasonable. To illustrate, the value could be nil when the condition is not met:

```
s length :: s controls† P†† val else 0;
s controls q :: s length lt (q length) excludes (s controls† q).
```

However, the new function precludes controlling infinite sequences. In particular, the length function could not now be defined as it is above (in terms of the new "controls").

We can, of course, define a function which has the value "s" when its length exceeds a particular number:

```
s controls q :: s .  (rid *) .  q;
s lengthge i :: i pos††† controls s length = i then s;
s controls q :: s controls (q lengthge (s length)).
```

The function will be in error when the condition is not met.

The interpretation chosen, however, which preserves the potential unboundedness of "q", is to consider the controlled sequence always "infinite". In particular, the sequence is augmented by an infinite cycle of nils:

- -

†See [7] for definition.

††See [8] for definition.

†††See [10] for definition.

s controls q :: s . (rid *) . (q conc† (nil *)).

This interpretation gives the appropriate value for **sub**†† as defined above.

5. Sequence generation

"Sequence generation" may now be made more precise, and the fundamental nature of the language design decision it represents more fully considered. No sequence is ever generated--evaluated, produced--unless it is necessary, by virtue of its requirement as the value of a program, or in the computation of a value in the program. When produced, only the successive elements needed are generated. For example, in the "n pos" [10] function, neither "**le** n **exs**" nor "P" need (or could!) ever be generated in their entirety before the coapplication occurs. Only the portions needed before the result terminates are necessarily generated. Also, this termination may not need to occur; in:

<3; 25> controls (999 pos) ≡ <1; 2>

only two elements from "999 pos" need ever be generated. The implementation techniques developed by Abrams for APL [AB]--"beating" and "dragging"--are essential to the implementation of this basis.

The traditional notion of generator may be used to illustrate how this could be implemented. A generator for a sequence consists of a set of **own** variables unique to each instance of the generator, an initialization function, and a function for pulsing the generator. The pulsed function returns the successive elements of the sequence generated--one per pulse. The generator may, of course, run out of elements; the pulse function must indicate termination, and its caller must check for the condition.

- -

†See [5] for definition.

††See [4] for definition.

Evaluation of a program in the basis can be thought of as a sequence of calls on generators. A generator for the loop operator is trivial; an **own** variable is initialized to the loop operand; the pulse function always returns it as its value. The coapply generator initializes each of its constituent operand generators (preserved in **own** variables); its pulse function is the application of the results of pulsing its constitutent operand generators (after checking for termination). The accumulate generator initializes one **own** variable to the initial value of the accumulator; pulsing consists of setting the accumulator **own** to the application of the accumulator with the result of pulsing its operand sequence generator (after checking for termination).

Unfortunately, although generators are used frequently in programs, only languages which include the more general control structure "coroutine" can be used to describe the generators in any clean way. The "universally understood" languages LISP, FORTRAN, Algol, APL, and SNOBOL are poorly designed for the expression of such structures. Even SIMULA [DN], which allows coroutines through "activities", cannot be used well to describe the generators for the basis because of its type structure. A further discussion of implementation considerations is left until Chapter IV.

The implications of generation in terms of the design criteria of naturalness and power revolve about the programmer's conceptualization of a task vs. its implementation. "Structured programs" have the property that they are hierarchically decomposable. Frequently, this gives rise to implementations incorporating "passes" over the data: an entire sequence is processed by a function producing a result sequence, which is passed on to the next function, etc., until the output sequence is finally produced. The notion of generation preserves this structured decomposition but allows it to include unbounded input and result sequences.

When implementing a program designed in passes over successive results, the tendency is to attempt to "collapse" the passes; "single pass" compiler considerations have become trite. In fact, the generative mechanism, and its coroutine implementation, are generally recognized as the efficient way to implement structured programs. The orthogonality between "structured programming" and "efficient programming" is at the heart of the "modularization" problem [PA] and is partially solved by the generative notion involved in the basis. We reiterate: programs may be hierarchically structured in passes, but the generative mechanism requires a coroutine implementation, automatically collapsing passes where possible.

6. Cosequencing

The fourth fundamental decision--to include the cosequencing operators "." and "/"--may be viewed as a combinatoric decision. The operators do combine data and programs to produce new data or programs, and are similar to the combinators in Backus' Red 3 language [BA,1972]. Such operators are significant to program expressiveness: in a very real sense, programmers do identify program elements with data elements and build control structures around them. This activity is generally masked by the subordination of data sequencing facilities to control sequencing facilities in most programming languages. Recently developed languages such as QA4 [RU] and PLANNER [HE] allow some identification of program elements with data elements, but then "hide" the data structure in a global data base. The use of set generative features in languages such as SETL [SC] also represents a limited form of program/data element identification. The use of generators in IPL-V [NE] permits cosequential identification; however, it shares the problems of LISP in that the generators have to be explicitly pulsed and produce results explicitly by "outputting" the generated elements. Thus, the cosequencing notions are at least skirted by extant languages as worthwhile programming features, not simply as combinatoric "tricks". The extent to which cosequencing operators may be developed for less homogeneous structures is the central subject of the remainder of this work.

7. Combinatorics

The decision to use combinatoric mechanisms is regarded as fundamental. Each of the primitive forms will ultimately influence the other's definition by how they interact in combination. As an example of such interaction note the effect of the decision to use a left-to-right evaluation sequence in combination with the accumulate operator. The accumulate operator must evaluate left-to-right, inasmuch as there is no rightmost element of an accumulation before its execution. Had we used a right-to-left evaluation scheme, the operation would have been entirely counterintuitive.

Combinatoric decisions influence and motivate the definitions of the relational and arithmetic operators. It was mentioned above that such definitions should be extended to include nil in their domain. This is done simply: all operations of type arithmetic and relational are nil when either argument is nil. Additionally, it is convenient to have undefined arithmetic operations--zero divide, overflow, underflow, etc.--produce **nil** as their value. This effectively imposes the interpretation on **nil** of an "undefined element", in the sense of "plus-or-minus infinity".

We may then write functions which can pretest their operands before applying the actual function. For example,

```
(n le i ge j) + 9
```

will have a non-empty value iff n is in the closed interval [i,j]. Its Algol counterpart:

```
if (n le i) and (n ge j) then n+9 else INFINITY
```

is less concise.

Combinations of conditionals and relationals provide further evidence of the conciseness gained by combinatoric devices. For example, the "max" function:

```
i max j :: i ge j else j
```

would require an extra clause in Algol:

```
max(i,j)= if i ge j then i else j.
```

From a language design viewpoint, combinatorics should not be "unnatural" in the following sense: when there is a clear choice between two possible interpretations for a construct and neither is clearly more intuitively appropriate, **the choice should not be made**. For example, it may be inappropriate to define addition between characters and integers, for there is no obviously appropriate choice for the result type.

This precept was violated in the choice of the left operand as the value of a true relation. We now replace that choice with more consistent interpretation based on the notion of "section" and an observation about of the usage of relations. We define the "minor" argument of a binary operation to be its right operand. Binary operators are considered to be instantiated in their minor argument when they stand unparenthesized:

```
a = b ≡ a (= b)
```

We modify the interpretation of relations:

```
a (rel b) ≡ if a rel b then a else nil
```

and

```
(a rel) b ≡ if a rel b then b else nil.
```

With this interpretation, the range specification can be made more naturally:

```
(i le) b le j
```

to have the value b when in the interval [i, j]. Most expressions involving relations involve arguments which are ranked, in the sense that one is varying and one is constant. The combinatoric decision represented by the above interpretation permits the more constant operand to be instantiated first as the minor operand. (Some formal work is required to assure the above interpretation is consistent; for example, in "a . (le *) . b", one must presume that the elements of "a" constitute the minor argument to the relational, because of the left-to-right evaluation sequence.)

Note also that it is particularly important that no flexibility is lost by including combinatorically useful interpretations of primitives. If one prefers to emphasize the symmetry of a boolean decision, for example, he can always revert to the boolean interpretation, as in:

```
a lt b
  then a
  else b
```

In one sense, combinatorics may be envisioned as maximizing the useful default interpretations of syntactic constructs, subject to the non-artibtrariness requirement mentioned above. We reiterate, the fundamental decision is to use combinatoric "power"--none of the particular decisions in these examples is neccessarily of global significance to language design.

Examples

The following examples are presented primarily to familiarize the reader with use of the basis in more realistic problems than have been presented above. The examples evoke extensibility issues which relate to claims of conciseness for the language. Throughout this work, each example program characteristically includes two sets of functions which will be referred to as "basic" and "ad hoc". The former are those functions which would presumably be part of a language developed from the basis. Although each is expressed in terms of the basis, the function may be either implemented or part of a library facility in the actual language. The "ad hoc" functions are very particularly related to the problem at hand, and could rarely be used elsewhere. Naturally, it is to the language's credit if the "ad hoc" functions for any particular task are few. That is, languages, once defined, are only ever rendered more concise through extension; hence, the ability to easily define functions for general usage is important.

Complex, inconsistent libraries can arise in any language; in the basis, care must be exercised not to terminate sequences in an ad hoc fashion and not to take the **val** until after the sequence has been isolated as a unique function. For example, the largest power of "2" less than or equal to a number "n" could be written:

```
lp2le n :: 1 / (mul 2 *) while† (le n) val
```

The following factorization would be preferable, however:

```
powersof i :: <1; 1/(mul i *) gen>;
powersof2 :: powersof 2;
powersof2le n :: powersof 2 while (le n);
lp2le n :: powersof2le n val;
```

for each of the components of the function is of potential utility in other contexts. Naturally the user must believe the original implementation will result from substitution rather than an actual layer of "generator calls".

- -

†See [9] for definition.

1. Matrix multiplication

In the following, "u" and "v" represent row vectors, "M" and "N", matrices (sequences of row vectors). All the functions in this example are basic:

```
M column i :: M . (sub i *);
        ≡ <M₁ sub i; M₂ sub i; ...>.
M transpose :: M column * . P†;
        ≡ < M column 1; M column 2; ...>
u rowmul v :: u . (mul *) . v;
        ≡ <u₁ mul v₁; u₂ mul v₂; ...>;
u sigma :: 0 / (+* . u) val;
        ≡ <0+u₁; 0+u₁+u₂; 0+u₁+u₂+u₃; ...> val;
u ip v :: u rowmul v sigma;
        ≡ 0+(u₁ mul v₁)+(u₂ mul v₂)+ ... + (uₙ mul vₙ);
r rM M :: r ip * . (M transpose)
        ≡ <r ip (M column 1); r ip (M column 2); ...>
M MM N :: M . (rM N *);
        ≡ <M₁ rM N; M₂ rM N; ...>
```

Although the basic functions are self explanatory, some problems are encountered in dealing with the potentially infinite transpose function. The transpose defined is appropriate for arrays with rows of unbounded length. Such an array could arise in a histogram for a set of system parameters in an operating system, for example, where termination of a row is tantamount to the system crashing, an event of finite but unbounded length. The transpose might then be a very useful function for a printer output routine.

However, the transpose function (by itself) will never terminate: its structure is simply an infinite loop, and the later rows of the transpose will consist of all nil elements, as "i" in "column i" exceeds the row length. Although we are dealing with the same phenomenon as encountered in insuring that the head function could deal with an infinite sequence, the termination is somewhat more complex. There are two issues

†See [8] for definition.

which complicate the function. First, we prefer to allow the transpose of arrays with different size rows. Second, we wish to allow empty elements in the array.

If we allowed empty elements only as the elements generated when the row was exhausted, we could safely terminate the array with the following function, which simply tests to see if the entire row consists of nil elements:

```
v notempties :: v .  (txs *) val then v;
M transpose :: M column *.  P while notempties
```

For example, if M = <<3; 9>; <1>; <4; 7; 8>>, then

```
M column * .  P
    ≡ <<3; 1; 4>; <9; nil; 7>; <nil; nil; 8>; <nil; nil; nil>; ...>.
```

Also,

```
<3; 1; 4> notempties
  ≡ <3 txs; 1 txs; 4 txs> val then <3; 1; 4>
  ≡ <3> val then <3; 1; 4>
  ≡ 3 then <3; 1; 4>
  ≡ <3; 1; 4>.
```

Similarly,

```
<9; nil; 7> notempties
  ≡ <9> val then <9; nil; 7>
  ≡ <9; nil; 7>
```

and

```
<nil; nil; 8> notempties
  ≡ <nil; nil; 8> val then <nil; nil; 8>
  ≡ <nil; nil; 8>.
```

However,

```
<nil; nil; nil> notempties
   ≡ <nil txs; nil txs; nil txs> val then <nil; nil; nil>
   ≡ <nil; nil; nil> val then <nil; nil; nil>
   ≡ nil.
```

Hence, the transpose terminates with:

```
<<3; 1; 4>; <9; nil; 7>; <nil; nil; 7>>.
```

Now note that the above function works for unbounded arrays: the transpose terminates iff the original array generator does, and the transpose generation proceeds along with (cosequentially with) the original array generation. However, the general case--permitting empty elements anywhere within the array--remains problematic. For example,

```
<<3; nil; 4>; <1>; <9; nil; 7>> transpose
   ≡ <<3; 1; 9>; <nil; nil; nil>; <4; nil; 7>>,
```

but the above function will terminate with the first element.

Two solutions suggest themselves. The simpler is to replace nil by our own version of "NIL"--a token. The normal problems with finding an unique element are attendant with this solution†, but it is of some pedagogic interest to illustrate how it may be accomplished. Assume, "NIL" is a unique element which cannot occur within the argument matrix. We may replace nil in an array by NIL using:

```
M fromnil :: M .  (.  (else NIL *) *).
```

For example:

- -

†Ultimately, an unbounded sequence of unique tokens in nested contexts is required.

```
<<3; nil; 4>; <1>; <9; nil; 7>> fromnil
   ≡ <<3; nil; 4>.(else NIL *); <1>.(else NIL> *);
        <9; nil; 7> .  (else NIL *)>
   ≡ <<3 else NIL; nil else NIL; 4 else NIL>; <1 else NIL>;
        <9 else NIL; nil else NIL; 7 else NIL>>
   ≡ <<3; NIL; 4>; <1>; <9; NIL; 7>>.
```

We may then remove NIL after applying the transpose above, using a similar function:

```
M fromNIL :: M .  (.  ( ne NIL *) *),
```

and can define the new transpose in terms of the old:

```
M transpose :: M fromnil transpose fromNIL.
```

However, finding a unique element not present in any array requires a dynamic unique name generation scheme which we are not prepared to deal with presently, and which we are never prepared to deal with efficiently. Thus, a second solution is proposed.

Again, we use the transpose function which terminates with a row of empty elements. The method used is to simulate a "boolean array" which has nonempty elements wherever the transposed array has elements of any sort. This boolean mask array may be generated:

```
M Bmask :: M .  (controls† (1*) *).
```

E.g.,

```
<<3; nil; 4>; <1>; <9; nil; 7>> Bmask
   ≡ <<3; nil; 4> controls (1*); <1> controls (1*);
        <9; nil; 7> controls (1*)>
   ≡ <<1; 1; 1>; <1>; <1; 1; 1>>
```

†See [7] for definition.

Applying the transpose above to this array will then terminate after the same number of rows are generated as in the required transpose for "M". Hence, this can control the length of the actual transpose:

M transpose :: M Bmask transpose controls (M column *. P).

This is the general transpose for 2-dimensional arrays which permits:

1. nil elements;
2. Rows of arbitrary lengths;
3. Arbitrary elements;
4. Unbounded cosequential generation of the transpose as the argument array is generated.

Another negative aspect of the functions is that many are primitive in APL, and hence, the basis is less concise for this problem. This is to be expected throughout: APL will always do better when problems are formulated directly in its representation. On the positive side, the transpose is more general than APL's (for matrices), in the sense that it allows unbounded length rows in non-homogeneous arrays with possibly empty elements.

Also note the ease with which the basis is extended; each of the defined functions is useful for a large class of problems. The same functions in Algol-like languages involve temporary arrays and loops, and by no means lend themselves to simple functional composition, as do these.

2. Recursive programs

Although any Turing Machine may be defined in terms of the basis†, and hence, all recursive functions are computable using it, such justification is not germane to

---

†See Appendix V for a construction.

higher-level language design. Recursion must be part of the basis unless suitable operators exist which allow a more concise specification of what would normally be accomplished through recursion.

We illustrate conversion of a particular recursive schema to the basis, and consequently to an iterative algorithm:

f(x) = if x = k **then** y **else** up(f(down(x)),x),

where "up" and "down" are not recursive functions. (The simple termination predicate is chosen to avoid clouding the primary issues involved; an arbitrary predicate and termination function may be substituted for (=k) and y, respectively, with only minor modifications to the function to be presented.)

The essential implementation device is to compute an accumulation sequence of values of the function "down". In an implementation this sequence would correspond to the "stack". The sequence, reversed, becomes an argument to an accumulation of the function "up". In particular, note that computation of the stack sequence preceeds the computation of f:

<down(x); down(down(x)); down(down(down(x))); ...>

until its last element is such that "down(lastel) = k". Substituting:

$<s_1; s_2; \ldots ; s_n>$

for the above sequence, the sequence:

$<up(y,s_n); up(up(y,s_n),s_{n-1}); \ldots$
$up(up(\ldots up(y,s_n),s_{n-1}),\ldots),s_1)>$

will be computed. The val of this sequence is the value of the function.

Hence, we can define a stack sequence function:

```
    x computestack :: <x; x/(down *) gen> while (ne k).
```

The recursion sequence reverses the stack and simply accumulates the recursive result:

```
    x f :: y/(up * .  (x computestack reverse)) val
```

where the reverse function is:

```
    s prefix y :: <y> conc s;
    s reverse :: <>/ (prefix * .  s) val.
```

Of course, this is less concise than recursion! However, recursion is frequently a poor way to implement a function, as the following example will illustrate.

To compute the exponential function "i" to the "jth" power, a recursive function (for constant i) is:

```
    f(j) = if j=0 then 1 else
               if odd(j) then i*f(j-1)
                   else f(j/2)*f(j/2).
```

We may write:

```
    k odd :: k mod 2 = 1;
    j down :: j odd then (j-1) else (j div 2);
    m up j :: j odd then i else m mul m.
```

Substituting these functions in the above schema then gives the appropriate function in terms of the basis.

Note that the recursive function may be rewritten:

```
    f(j) = if j=0 then 1 else
               (if odd(j) then i else 1) * f(j/2)*f(j/2)
```

where "/" indicates integer division. (This is a result of the fact that j-1 will be even when j is odd, and hence, that f(j/2)=f((j-1)/2) will be computed immediately on the next recursive call.)

We may now write:

```
j down :: j div 2;
m up j :: m mul m mul (j odd then i else 1)
```

Note that for this value of down:

```
x computestack :: <x; x/(div 2 *) gen> while (ne 0)
```

is the sequence of numbers arrived at by right-shifting an accumulator initially containing x (j in this case) on conventional machines. (Its reverse requires coupled accumulators and/or special instructions for efficient machine implementation.) The point is that from a canonical reformulaton of a recursive function, consideration of the resultant stack sequence may indicate a much more efficient implementation than would be expected; in this case, the standard "2 $\log_2$ j" multiplications algorithm is illuminated [KN]†.

The complexity of the above material perhaps indicates why "factorial" is normally chosen as the showcase recursive function. If one recognizes:

```
<n; n/(-1 *)> while (ne 0) reverse
```

as equivalent to:

```
n pos
```

the normal basis expression for factorial will be derived using the above schema substitution:

†See Appendix IV for a similar example (right-to-left instead of left-to-right) and its compilation from the basis.

n factorial :: 1/(mul * . (n pos)) val.

Reexamination of Decisions

The fundamental decisions involved in the development of the basis are:

1. To use an applicative language framework;
2. To apply the gotoless constructs to nested sequential structures;
3. To use a generative technique for the production of sequences;
4. To include non-trivial "apply" operators as primitive--in particular, "." and "/";
5. To use a syntax with high combinatoric power.

Of remaining interest regarding the first two decisions is the question of how the use of pointers so constrained differs from traditional use of reference variables and assignment statements. Of particular interest is whether additional control constructs are needed to accomplish what are considered correct, well-controlled notions that the removal of the pointer from data structures precludes.

The decision to use a generative mechanism is based on the separate views programmers have of the action of algorithms on data structures and the way the action actually occurs. The generative notion is presently quite simplistic and extremely constrained. What remains to be seen is the extent to which generative aspects of programs may exist in the constrained environment of an applicative language. The implementation technique of coroutine usage for generative programming activities is clearly the fundamental mechanism to be studied: how can we include the coroutine and how must it be constrained to fit the structured programming framework? The notion of factored termination is separable from that of generation; one can imagine a generative specification of APL operations. In fact, Abram's work [AB] essentially employs a generative implementation about which efficiency considerations are more easily made. However, there is no way for the programmer to exploit a generative specification or implementation of APL, for all operations are defined over finite arrays. Additionally, APL's consistent extension to deal with unbounded sequences in any general way would be non-trivial. By separating length-controlling facilities for sequence generation from their specification, the issue of generation of results becomes fundamental in the basis--programs may be written with infinite sequences in mind.

Choice of the cosequential operations over a recursive generation base represents another major decision, largely independent of the others. The extent to which the cosequential application of programs to data to produce a result sequence summarizes operations over homogeneous sequences remains to be presented, as well as the extent to which "corecursive" operations over non-homogeneous nested sequential structures can be developed.

The major implication of this section lies in the development of a syntax with high "combinatoric power", not in the particular sysntactic mechanisms used in examples. The combinatoric power relates to extension facilities as well as to primitive operation definitions. The decisions to use an operator language, a typeless language, a nonstandard definition of the relationals, to apply the gotoless constructs as operators, to rely on the "partially instantiated function", etc., all relate to concise combination of operators. In fact, the cosequencing operators can be looked upon as "combining mechanisms". In order to separate issues of syntax from semantics, Backus' work [BA,1972,1973] encourages us to look upon the entire activity of programming as an exercise in combinatorics. Although we do not hold this view (our operators are derived from machine-oriented operations), Backus' work makes it clear that combinatoric considerations are important to language design. Insofar as possible, we wish to avoid syntactic issues for the remainder of this work, in order to concentrate on the semantic issues of cosequencing and its relation to traditional data/control structures.

CHAPTER III

RECURSION

The initial language basis adequately summarizes many sequential operations on data structures and the sequential creation of many homogeneous data structures. In this chapter, non-homogeneous program and data structures are examined in an attempt to describe more complex operations concisely.

It is important to notice that the basis at this point is computationally complete, for the effect of a Turing Machine may be encoded easily (see Appendix V). Hence, hereafter, the addition of operators to the basis represents efforts to obtain "difficult" effects more easily. The approach is guided by common usage of traditional language facilities not present in the basis, in conjunction with cosequential-generation notions introduced previously; no attempt is made to duplicate the traditional facilities themselves.

The primitives added hereafter must be considered even more tentative than those in the initial basis, for more complex entitites are considered and combinatoric/involutional problems expand exponentially. Additionally, issues orthogonal to sequential considerations--viz. association mechanisms, atomic representation issues, types, etc.--complicate on the more realistic problems arising from non-homogeneous sequences. However, the extent to which they are bothersome is reduced by careful selection of examples for presentation; i.e., the reader is led down a "primrose path" in order to amplify the relevant issues.

We wish to deal with a broader class of nested-sequential structures than can be handled easily with the initial basis. In particular, in the initial basis sequences consisting of elements which may be either sequences or primitives (non-homogeneous sequences) must be dealt with via explicit "pulsing" of the data structures by the program. The use of "car" and "cdr" in LISP constitute "pulsing" a list--see Chapter I, for example. Analogous functions must be defined to deal with recursive structures in the language basis (see Appendix VII for an example). It is exactly this pulsing which is is eliminated for sequential structures by the cosequencing operators, and so we look for cosequential recursive ("corecursive") operators to deal with non-homogeneous sequences.

The task of compiler construction is considered as a motivation for corecursive operations. A fairly standard breakdown of this task consists of functions--"lexemes, syntax, code, output"--which, when applied successively to the input sequence produce the translated program, viz.

inputseq lexemes syntax code output ≡ compiled program.

Commonly, the above program would be called a four pass compiler--one pass per function applied to the argument†.

The implementation of a compiler as a single pass over the input data is of frequent concern to compiler writers. In such an endeavor, the breakdown above is retained, but the functions are executed "cosequentially". In particular, assume "syntax" creates a parse "tree" (in terms of "atoms" from the function "lexemes"), and "code" (optimization) proceeds by considering successive "syntactic handles" of the tree [FG]. The program starts by attempting to "output" code. Initially, no code has been produced, so "output" pulses "code" until it produces enough of a tree--a statement, for example--for which "output" can output code. Naturally, "code" has no input initially so it calls "syntax" until it can produce enough handles for code to optimize. This process continues until enough of the input sequence is read to produce enough lexemes that "syntax" can generate enough tree for "code" to proceed with satisfying "output". The process then cycles. That this process is "cosequential" is clear; however, the structures with which we are dealing--trees--cannot be handled well with the primitives from the initial basis. In what follows we shall exhibit cosequential operators for trees.

Recursion

In order to study recursive program and data structures, a recursive definition ability is added to the basis (temporarily). The gains represented by the cosequencing operators of the initial basis are then considered briefly in terms of a recursive formulation of these operators. It will be shown that the notions of cosequencing, unbounded generation and factored termination affect such a formulation.

Although it would be possible to extend these notions to recursion--and include the recursive definition capability permanently--we choose to limit the recursive structures definable within the basis. This limitation parallels the way the gotoless constructs limit the sequential structures definable in programming languages that lack the goto. Hence,

†Historically, the notion of "pass" referred to the number of times the program had to be read by the compiler until ultimately enough information was available to compile it. Present day (large memory) machines have modified this to the number of complete scans of intermediate representations the program undergoes during the compilation.

the existence of "recursionless" constructs, analogous to gotoless constructs, is postulated and studied in detail below.

We choose to (temporarily) install the recursive definition ability via introduction of a quote operator, "'", which precludes operator body variable binding at operation definition time. For example,

```
x f :: x = 1 else (x-1 'f mul x)                                [1]
```

then represents the factorial function expresed recursively (divergent for 0). Note that if "f" were not quoted in the body, whatever previous definition of "f" exists would be the appropriate reference. (Quoted names have the "normal" LISP binding, names which are not quoted have the "FUNARG" [McC,1965] or Algol binding.)

To illustrate recursive definitions of "." and "/" we need two primitive functions, "1st" and "tail", analogous to the LISP functions "car" and "cdr", respectively, when applied to lists.

```
x id :: x;
x null :: <> gen;
list 1st :: list .  <id> val;
list tail :: list .  <null; id * gen>;
```

Given these two functions as primitive, we can describe "." and "/" (used as names below) recursively:

```
s . q :: <s 1st (q 1st); s tail '. (q tail) gen>;              [2]
v / s :: <v (s 1st); v (s 1st) '/ (s tail) gen>
```

This implementaion is inadequate in that no termination conditions are present. With an additional primitive, "isemptyseq", the functions could terminate when "s" or "q" is empty. By providing a mechanism for dealing with "quoted programs" (a desirable provision) we could terminate contingent on escape function values. However, this would still be inadequate, for unbounded sequences such as "+*.P" would never terminate.

Providing a truly adequate evaluation mechanism is tantamount to implementing the basis (considered in Chapter IV and Appendix IV). Essentially, a recursive coroutine simulation is required; more arguments are required to each function call of "." and "/".

Hence, a recursive implementation requires a significant amount of detail to simulate "." and "/", and is less "natural" than the basis for modelling cosequential activity.

As we mentioned above, the ideas of cosequentiality and unbounded generation can be extended directly to recursive functions such as [2]; we do so in a more constrained manner below. The above discussion simply points out that it is no easier to implement the cosequencing operators in the framework of a recursive language than in the framework of a sequential language.

Notice, also, the use of "1st" and "tail" constitutes explicit "pulsing" of a data structure by a program structure. For some programs in the basis, this explicit pulsing is eliminated by the "." and "/" operators. Programs written in the basis in which pulsing is required may be characterized as either "recursive" or "codependent" (or both). Two structures are "codependent" if they depend on each other functionally but neither can be classified as functionally superior in importance. An "inventory file" and a "manufacturing order file" exhibit signs of codependence. The necessity for such structures arises when a function in an applicative language has no "clean" functional decomposition. Such structures are considered in Chapter IV.

Recursive structures are characterized by potentially infinite nesting, as well as some degree of "branchiness" (see "essential recursion" below). This infinite nesting could arise either from (traditional) explicit recursive use of finite operators or from use of potentially infinitely recursive operators analogous to "." and "/". The simple extension described above handles the first case. For the second case, we will define recursive analogs of "." and "/". These will have the properties of the corresponding operators in the initial basis, permitting:

1. Unbounded cosequencing, and hence,
2. Factored and implicit termination, and
3. Factored data/program representation (non-pulsed data structures).

This chapter represents a search for recursive operations which are amenable to the ideas of cosequentiality, unbounded generation and factored termination introduced in the initial basis. In one sense, such a search is premature: the initial basis relies heavily on the ability of the gotoless constructs to adequately summarize sequential activity of programs. This point of departure is significantly more advanced than that for recursion. We do not possess a set of "recursionless constructs" which adequately cover recursive program activity in the same sense as the gotoless constructs cover sequential program activity.

Intellectually, this parallel between the goto and the recursive call is very appealing--multiply nested mutually recursive functions are easily as complex as programs with "rats' nests" of goto statements†. Ideally, the definition of a set of recursionless constructs would lead to the elimination of a recursive function definition facility in programming languages. Naturally, this elimination would complete the removal of complex program-to-program pointers from programs, relegating function call to a substitution process (which is is already required of any practical implementation of the initial basis).

Thus, in what follows, we are faced with two separate issues: defining "recursionless" constructs and understanding the relationship of of the concepts of the initial basis to these constructs. Unfortunately, our results in this area are only "suggestive"--we cannot propose a set of recursive operators which cover well all instances of recursion (again, in the same sense as gotoless constructs cover well all instances of sequentiality). It is not surprising that our results in this area are incomplete, in view of the history of the development of the gotoless constructs, the difficulty of expressing their operators in the basis, the constraints imposed by requiring that they apply to data structures as well as program structures, and the unbounded generation notions. However, we do feel that a search for "recursionless" constructs is a fruitful area for future research, and attempt to indicate an approach to this problem in the following pages.

Sequential Functions on Recursive Structures

Two very common types of recursive algorithm are considered in initial attempts to define recursionless constructs for the evaluation of functions on recursive structures: "top down" and "bottom up". From the recursionless constructs which arise from these considerations, operators are derived for inclusion in the basis. The operators so derived, although arising from quite specific recursive algorithm types, are extremely general when used in combination. Hence, at the end of this section they are related to some very general recursive forms.

1. Top down recursionless construct

Although "top down algorithms" are very common, languages do not generally contain constructs which permit their concise specification or explicit identification as such. When one refers to a "top down" evaluation procedure defined on a tree (arbitrarily nested sequence, in this context), some function is applied to the root node

- -

†Arbitrarily complex programs using gotos may be simulated with procedure calls [vW,1966].

before any function is applied to its subnodes. This may be expressed recursively in the basis as:

```
  t topdown :: t td . ('topdown *)                                    [3]
```

where "td" is the top down function applied. For example, if

```
      t td :: <t 1st; t val>
```

then the tree argument ("t") is simply trimmed by the top down function:

```
      <<1;2;3>; 4; <5;6>> topdown
        ≡ <<1;2;3>; 4; <5;6>> td . (†topdown *)
        ≡ <<1;2;3>; <5;6>> . (topdown *)
        ≡ <<1;2;3> topdown; <5;6> topdown>
        ≡ <<1;2;3> td . (topdown *); <5;6> td . (topdown *)>
        ≡ <<1;3> . (topdown *); <5;6> . (topdown *)>
        ≡ ...
```

This example will not terminate (except perhaps from an error of attempting to coapply an integer to a sequence).

Hence, some termination predicate ("tp") is used to decide whether to continue the recursive algorithm, and a termination function ("tf") is applied when the recursion halts along any path in the structure; i.e.,

```
  x topdown :: x tp then (x tf) else (x td . ('topdown *))            [4]
```

Of course, this is a very simplified version of a topdown algorithm, for there may be multiple arguments, mutually recursive topdown functions, etc., and, in fact, normally some information is passed down as recursion occurs. These will be considered in some detail later in this section. However, the relationship of the recursive control to the invoked function is captured by this formulation.

Normally, the notion of a "top down" algorithm implies that the result of the top down function itself is related to the original node; i.e., "td" acts as a selector of a sequence of subnodes. By not insisting on the selector relationship, the functions

- -

†At this point we may drop the quote ("'"), for the function "topdown" (in this case) is now defined.

above ([3] and [4]) are, in fact, too general to represent top down functions constrained to trees. This generality is convenient, for it provides a broader notion of top down algorithm which is constrained not to trees, but rather to a control structure which is a tree. This models the intuitive notion of top down algorithm very precisely. If this tree-like control strucure arises from application of a top down algorithm to a tree, the nodes of the control structure will correspond to the nodes of the argument tree. A notion of "corecursion" is emerging.

Despite the objections to the form [4], we can attempt to define a recursionless construct for an Algol-like, higher-level language. Its syntax might appear thus:

```
<recursionless construct> ::=                                   [5]
   topdown <control variable>←<initial value>
   <termination specification>
   do <top down function body>;

<termination specification> ::= <termination part> <application part>

<termination part> ::= <empty>
                     / until <termination predicate>
                     / while <termination predicate>

<application part> ::= <empty>
                     / whence <termination function>
```

where the correspondences: "tp" with <termination predicate>, "tf" with <termination function>, "td" with <top down function body>, are only approximate. (Empty alternatives for the <termination part> are to permit a default interpretation for these constructs.)

Although semantics of such a construct would be very language dependent, the construct itself is to be included in an expression language, and the language must be equipped to deal with lists. Then the value of the <top down function body> must be a list. Recursion will occur on each element of the list produced by the <top down function body>. (These elements are normally themselves lists.) The control variable takes on the value of the current subnode at any point in the recursion, and the variable is available for reference in the various parts of the construct. Before recursion occurs on any node in the structure, the termination predicate is tested. When satisfied, the action of the termination function takes place; otherwise, recursion occurs. For example, a function which adds "3" to the terminal nodes of a tree, "T", might be written:

```
topdown t←T until atom(t) whence t←t+3 do t;                    [6]
```

A more complex function, which does the same only to nodes arising from the first and third branches from each node, could be written:

```
topdown t←T until atom(t) whence t←t+3                          [7]
    do concatenate (t[1],t[3]).
```

The nature of the created value would be particularly language dependent--i.e. the construct might be modifying a recursive data structure as intended in [6] and [7], or it might be creating a new one. Several constructs may be necessary to cover these interpretations.

At this point we have a recursionless construct; before proceeding to the development of others we consider the problems involved in putting such a construct into the basis. Clearly, we would not include it directly, but would rather reformulate it as separate operators dealing with the various portions of this somewhat command-oriented syntactic construct.

In order to do so, we examine the loop construct of Algol-like languages and note the aspects which allowed its factorization and subsequent inclusion in the basis. A traditional predicate based construct, the **for** statement, consists principally of:

```
for <variable>←<exp> [until/while] <termination predicate>
  do <loop body>.
```

This construct takes on several different forms in the basis depending on whether the variable is used to count, accumulate, or index, but invariably the termination predicate is separate from the loop in another (cosequential) loop in which one of the escape operators is used, or else the termination is implicit.

Naturally, we would like to retain the separation of termination from a recursive top down operator. For the moment, we ignore the various relations of control variable to top down function†. We can then consider including a top down operator in the basis, "↓", which will accomplish the top down recursion between a tree "t" and right-unary function, "td":

```
t ↓ td :: t td . ('↓ td *)                                      [8]
```

A natural implicit termination condition is that recursion has reached a primitive node (integer, character, etc.).

However, we are then left with the problem of the <termination function>. If we default this to the identity function (again temporarily), we can then write some reasonable top down functions. For example, using the "tail" function defined above, we have:

```
<1; <3;5;7>; 9> ↓ tail
  ≡ <1; <3;5;7>; 9> tail . (↓ tail*)                                      [9]
  ≡ <<3;5;7>; 9> .  (↓ tail *)
  ≡ <<3;5;7> ↓ tail; 9 ↓ tail>
  ≡ <<3;5;7> tail .  (↓ tail *); 9>
  ≡ <<5;7> .  (↓ tail *); 9>
  ≡ <<5 ↓ tail; 7 ↓ tail>; 9>
  ≡ <<5;7>; 9>
```

which is the tree with all initial sub-nodes removed.

A more complex function which selects only the odd sub-nodes at each node is:

```
t oddsn :: t ↓ (.(<id; null> gen *))
```

For example,

```
<1;<2>; <3; 4; 5>; 6; <7>> oddsn
  ≡ <1; <2>; <3;4;5>; 6; <7>> .  <id; null; id; null...>.  (oddsn *)
  ≡ <1; <3;4;5>; <7>> .  (oddsn *)
  ≡ <1 oddsn; <3; 4; 5> oddsn; <7> oddsn>
  ≡ <1; <3;4;5>.<id; null; id; ...>.(oddsn *);
    <7>.<id; null...>.(oddsn *)>
  ≡ <1; <3;5> .  (oddsn *); <7> .  (oddsn *)>
  ≡ <1; <3 oddsn; 5 oddsn>; <7 oddsn>>
  ≡ <1; <3; 5>; <7>>
```

- -

†Throughout this section, the top down operator is generalized (through redefinition) to enable effects which would be obtained through the use of a control variable. Do not be misled by the constrained nature of the first few definitions of the top down operator.

Hence, we have a viable implicit termination mechanism. The problem of application of the termination function (the "whence" clause in [5]) can be surmounted to some extent by the introduction into the basis of primitive predicates:

x **atom?** ≡ x if x is atomic (primitive), **nil** otherwise;
x **seq?** ≡ x if x is a sequence, **nil** otherwise.

If it is desired to apply function "tf" to the implicitly terminal nodes, we can rewrite the top down function to do this. For example, in the tree with initial nodes removed [9], if it is desirable to add one to the terminal nodes, we may write:

x auxf :: x **atom?** +1 **else** x

Then,

```
<1; <3;5;7>; 9> ↓ (tail . (auxf *))                                  [10]
  ≡ <1; <3;5;7>; 9> tail. (auxf *). (↓ (tail . (auxf *)) *)
  ≡ <<3;5;7>; 9> . (auxf *) . (↓ (tail . (auxf *)) *)
  ≡ <<3;5;7> auxf; 9 auxf> . (↓ (tail . (auxf *)) *)
  ≡ <<3;5;7>; 10> . (↓ (tail . (auxf *)) *)
  ≡ <<3;5;7> ↓ (tail . (auxf *)); 10 ↓ (tail . (auxf *))>
  ≡ <<3;5;7> tail . (auxf *) . (↓ (tail . (auxf *)) *); 10>
  ≡ <<5;7> . (auxf *) . (↓ (tail . (auxf *)) *); 10>
  ≡ <<5 auxf; 7 auxf> . (↓ (tail . (auxf *)) *); 10>
  ≡ <<6;8> . (↓ (tail . (auxf *)) *); 10>
  ≡ <<6 ↓ (tail . (auxf *)); 8 ↓ (tail . (auxf *))>; 10>
  ≡ <<6;8>; 10>
```

However, this skirts the termination function issue to a large extent. . Both the termination predicate and the termination function are troublesome. In the following section we generalize the top down function somewhat, and approach the problem of termination in greater detail, for the issues involved have analogs in the initial basis and are related to the coroutine notions of Chapter IV.

2. Top down reexamined

We may summarize the problems with our attempts to express the top down mechanism of the preceding section as operators in the basis:

1. Termination is necessarily implicit;
2. The termination function had to be applied from within the top down function itself;
3. The termination function could not itself produce a sequence--otherwise, the top down mechanism would have continued to be invoked.

Additional problems remain, related principally to the <variable> portion of the recursionless construct specification. In the gotoless loop constructs, several constructs were required to replace the control variable--the use of "." and "/" as well as some functional tricks frequently applied to the positive integers, "P". These problems occur within top down control, also, although they may be subverted through encoding tricks to a large extent.

To illustrate, a top down function, "f", which replaces each terminal node ("n") of a tree "t" with the length of the longest sequence of which "n" is a subsequence is programmed below†:

```
i max j :: i gt j else j;
i merge s :: s seq? then <i; s gen> else i;
t tdf :: t length - 1 max (t 1st) merge * .  (t tail);
t f :: <0; t gen> ↓ tdf
```

Clearly, the recursive implementation is more concise and intuitive:

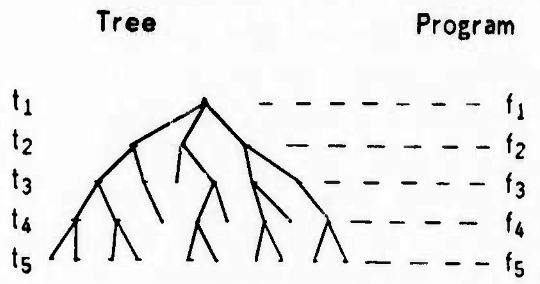
```
t aux len :: t atom? then len                                    [11]
               else (t .  ('aux (t length max len) *));
t f :: t aux 0
```

The failing is a natural one common to recursive functions restricted to a single argument--secondary arguments must be encoded. In the material below, top down functions for more than one argument are developed. Also, some of the objections to the previous top down function are removed.

- -

†That is, with the degree of the node of highest degree on the path from the root to "n".

Presently, a top down function is developed which represents a recursive function defined on a single argument but which is a simple extension to the previous top down function and which permits some multiple argument recursion effects quite easily. It essentially involves the notion that recursive control can be imposed on a sequence of functions instead of a single function, "td". A vertical relationship between the nodes of a tree and the elements of the program sequence is established; viz.



In particular, the top down function above [8] is **redefined** as:

```
t ↓ fs :: t (fs 1st) . ('↓ (fs tail) *)                                  [12]
```

Recursion terminates when the function sequence terminates or when t is atomic. The ability to terminate the sequence of sections ("partially instantiated functions") permits termination prior to the ad hoc nodes of the tree. Also, the depthwise orientation of the function sequence application allows some effects that would require either multiple arguments to the recursive function or mutually recursive functions. The former effect is illustrated below by a function which trims its argument tree by going no deeper than "n" levels, replacing the nodes at level "n+1" by the number 40000:

```
x lid y :: x;
x rid y :: y;
t depth n :: t ↓ <lid * .  (n pos) gen; rid 40000>
```

For example,

```
<1; <2; <3;4>; 5>; 6> depth 2
  ≡ <1; <2; <3;4>; 5>; 6> ↓ <lid * .  <1;2> gen; rid 40000>
  ≡ <1; <2; <3;4>; 5>; 6> ↓ <lid 1; lid 2; rid 40000>
  ≡ <1; <2; <3;4>; 5>; 6> (lid 1) .  (↓ <lid 2; rid 40000> *)
  ≡ <1 ↓ <lid 2; rid 40000>; <2;<3;4>;5> ↓ <lid 2; rid 40000>;
     6 ↓ <lid 2; rid 40000>>
  ≡ <1; <2;<3;4>;5> (lid 2).(↓ <rid 40000> *); 6>
  ≡ <1; <2 ↓ <rid 40000>; <3;4> ↓ <rid 4000>; 5 ↓ <rid 40000>>; 6>
≡ <1; <2; <3;4> rid 40000; 5>; 6>
≡ <1; <2; 40000; 5>; 6>
```

The recursive function which it replaces requires either two arguments (for fixed "n"), or use of the messy encoding scheme presented above in [11]:

```
t auxf d :: d gt n then 40000
                   else (t atom? else (t .  ('auxf (d+1)*)));
t depth :: t auxf 1
```

Hence, a very simple aspect of multiple argument recursive functions is captured by the (final) reformulated top down function [12].

A more important aspect of recursive functions permitted by the sequential/recursive top down function is the ability to define mutually recursive functions by alternating the operators applied in the top down sequence. For example, when dealing with "and/or" trees†, it is normally the case that different types of nodes are treated differently. Given such a tree, "aot" ("or" node at root), we can program a function which selects the first alternative consistently (at each "or" node) as:

```
aot basetree :: aot ↓ (<1st; id> ;;gen *)
```

A canonical recursive formulation would be best written using mutually recursive functions:

```
aot and :: aot atom? else (aot .  ('or *));
aot or :: aot atom? else (aot 1st and);
basetree :: or
```

Inasmuchas "td" is a sequence, we can consider the effect of allowing escape functions in the sequence. Although several choices for the meaning of an escape are possible (terminate recursion, terminate use of any successive elements of the function sequence throughout the remainder of the tree) the most reasonable seems to be to terminate the sequence along the current path only. That is, recursion is terminated for

- -

†And/or trees are frequently used in game-playing applications, syntax tree representations, theorem proving systems, etc., [NI] where problems can be formulated on the mututal occurrence (and) of choices from a set of alternatives (or).

the particular node in question, but for no others. For example, the function "trim" in:

```
s L p :: s length p then s;
t trim n :: t ↓ (L (le n) exs f *)
```

would apply "f" to each subtree until it reaches subtrees whose length (i.e. "order" or "node size") exceeds "n".

The value at the terminated node can be defined as the node which caused the escape in the case of **exs** and the function value (operand) of txs. Thus, the above example terminates recursion with the nodes whose length exceeds "n". The original recursionless construct [5] should probably not be extended to approximate the embellished top down operator, unless an analog were introduced for loops. In particular, one could imagine the **do** loop consisting of a sequence of loop bodies, successive elements of which are used as the loop is pulsed. Although the possibility should not be ruled out, such a construct seems a rather unlikely candidate for inclusion in most languages.

At this point there are three problems with the top down operator:

1. The terminal function ("tf" in [4]) must be applied by top down in an awkward manner (see example [10]);
2. Binary argument recursion requires encoding techniques (see example [11]);
3. We have not achieved a separation of termination from top down generation, and, in fact, are unable to terminate well other than with atomic nodes-- i.e. implicitly.

Presently, we discuss two more top down operators which alleviate problems 1 and 2, which further develop the notions of cosequential/recursive generation, and which relate very directly to the initial basis. Their development helps to illuminate the third problem.

3. Top down coapplication

To reiterate, the definition of the top down operator with termination conditions is:

```
t ↓ td :: td emptyseq then t else
              (t atom? else (t (td 1st) atom?)
              else (t (td 1st) .  ('↓ (td tail) * )))
```

In particular, we never recur on atomic nodes. This has the beneficial effect of not requiring the top down functions to be defined on both sequential and atomic arguments. However, it requires the messy implementation of [10] whenever we want to recur on atomic nodes or when we simply wish to apply a terminal function "tf", to the atomic nodes.

Notice that if we rewrite the top down form as though there were no implicit termination (other than not to recur on atoms produced by the "td" functions) we have:

```
t ↓ td :: t tp then (t tf)
               else (t (td 1st) atom?)
               else (t (td 1st) .  ('↓ (td tail) * ))
```

If "td" sequences are defaulted to the identity sequence, "id *", we have simply:

```
t ↓ (id *) ≡ t tp then (t tf)
                  else (t .  ('↓ (id *) *))
```

We could then consider merging the termination predicate and function, which seems quite consistent with the combinatoric nature of the basis--i.e., we could define an operator:

```
t ! tf :: t tf else (t .  ('! tf *))
```

This would terminate recursion when "tf" returned a non-empty value, with that value. Otherwise, recursion would proceed on each element of the argument sequence, "t". However, this eliminates nil terminal nodes and propagates the problem we are trying to eliminate: the "tf" function must then be defined on both sequences and atoms. Hence, an alternative formulation, applying the function "tf" only when the node is atomic may be considered:

```
t ! tf :: t atom? then (t tf)
                 else (t .  ('! tf *))
```

This does permit terminal function application to atomic nodes, which is one problem we intended to solve. In particular, example [10] may be rewritten:

```
t ↓ (tail *) ! (+1)
```

The extension of the above operator to allow a sequence of terminal functions as the right argument is consistent with "↓", and provides the same multiple argument/mutual recursion capabilities as it does in the original top down function. Hence, the top down "coapplication operator", "!", is defined:

```
t ! tf :: tf emptyseq then t else                                [13]
              (t atom? then (t (tf 1st)))
              else (t .  ('! (tf tail) * ))
```

(Throughout this chapter, the use of nil as an element is restricted because we are using the language itself to describe the effects we want in the language. Presumably an implementation would be more careful about such a restriction.)

For example, if we wish to replace all terminal nodes with their depths in the tree, we may write:

```
t ! (rid * .  P)
```

and, hence,

```
<1; <3;7>; 9> ! (rid * .  P)
  ≡ <1 rid 1; <3;7> ! <rid 2; rid 3; ...>; 9 rid 1>
  ≡ <1; <3 rid 2; 7 rid 2>; 1>
  ≡ <1; <2;2>; 1>
```

The notation "!" arises from the correspondence between this function and a "vertical" coapply operator; the cosequential correspondence between the depth of the argument tree and the section's index in the "tf" sequence is apparent.

4. Top down accumulation

The second problem we wish to solve is that of binary argument recursive functions. We do not solve it entirely here, but develop the top down accumulation operator, "∀", to relate one aspect of the multiple argument recursive function to the rest of the basis.

Frequently, top down functions act as accumulations, with the accumulation sequence branching recursively to the subnodes. For example, the function [11] (rewritten here as [14]) defines a tree, each terminal node of which is the length of the maximal sequence (node size) of which it is a subnode in tree "t":

```
t f maxel :: t atom? then maxel else                          [14]
               t .  ('f (t length max maxel)*);
t maxltree :: t f 0
```

The accumulated value is the maximum of the length and the accumulator--"maxel" in this case.

Notice that this effect cannot be obtained using the final top down function [12], for the right argument passed ("maxel") depends on both "t" and "maxel". In addition, the topdown recursionless construct above [5] does not permit a multiple variable capability. Before introducing an operator into the basis to accomplish this effect, we can consider extending [5] to obtain the effect in the traditional language "recursionless" cnstruct. An effective, simple method for allowing it would be to let local variables defined in the <top down function body> be propagated in the recursion. Initial values would have to be set in the declaration and the value retained when recurring. The above function could be written:

```
topdown T←t until atom(T) whence maxel
  do begin
          integer maxel = 0;
          maxel ← max (length(T), maxel);
          T
     end.
```

Naturally, this mechanism would also be very language dependent, and a separate phrase may be preferred for the accumulator specification and subsequent value.

The operator we are about to develop for obtaining this effect is related to the way multiple variables are handled in the initial basis with iterations. In particular, in the initial basis a form of accumulation which occurs frequently is:

```
iv / (fs .  s)
```

It occurs so frequently that it is reasonable to attempt to make an operator which depends only on "fs" and "s" which accomplishes the effect; e.g.,

```
fs reduce s :: fs functionzero / (fs .  s)
```

where the initial value depends on the function itself. Another possibility is to simply write the function and have its value be a section:

```
fs reduce s :: / (fs .  s)
```

Yet another is to insist that the function sequence have the initial value as its first element:

```
fs reduce s :: fs 1st / (fs tail .  s)
```

Any of these operators could enter the basis, or could perhaps replace the accumulation operator of the basis. The top down accumulation operator is an analog of the last choice:

```
t ∀ td :: t atom? else (td .  <id;id> length = 1)
                  then (td 1st)
                  else (t .  ('∀ <t (td 2nd) (td 1st); td drop 2> *)
```

where

```
s drop i :: s .  <i head (null *) gen; id * gen>
```

The accumulation takes a tree argument, "t", and a sequence of binary operators, "td", preceded by the initial value of the accumulator. These functions are applied to the non-terminal nodes and the current value of the accumulator. When the function sequence terminates or the tree node is atomic, the accumulator replaces the terminal node in the result.

The function defined above [14] may then be written:

```
t maxltree :: t ∀ <0; length max * gen>
```

For example,

```
<1; <2;3;4;<5;6>>; 7> maxltree
  ≡ <1; <2;3;4;<5;6>>; 7> .  (∀ < <1; <2;3;4;<5;6>>; 7> length max 0;
                                        length max * gen> *)
  ≡ <1 ∀ <3; length max * gen>;
       <2;3;4;<5;6>> ∀ <3; length max * gen>;
       7 ∀ <3; length max * gen>>
  ≡ <3; <2;3;4;<5;6>> .  (∀ <<2;3;4;<5;6>> length max 3;
         length max * gen>*);
       3>
  ≡ <3; <2 ∀ <4; length max * gen>; 3 ∀ ...; 4∀ ...; <5;6> ∀ ...>; 3>
≡ <3; <4;4;4; <5;6> . (∀ <<5;6> length max 4; length max * gen>*)>; 3>
≡ <3; <4;4;4; <5 ∀ <4; length max * gen>;
        6 ∀ <4; length max * gen>>>; 3>
≡ <3; <4;4;4; <4;4>>; 3>
```

The accumulation operator is the most tentative of the top down operators introduced, for the operation never applies to terminal nodes and the association of the initial value with the function sequence is distasteful. However, it is clear that a relationship exists between cosequential accumulation and top down accumulation, and, in fact, the basis should ultimately be reformulated to emphasize this consistency. There is a similar relationship between "t ↓ td" and "iv / fs". More extensive study along these lines is necessary before a concrete reformulation can be made.

5. Summary of top down operators

At this point a top down "recursionless" construct has been sketched for use in traditional Algol-like languages. The construct arises from pragmatic considerations of how recursion is frequently used to generate a recursive structure. From this construct (and imagined extensions) several top down operators--"↓, !, and ∀"--have been defined, which have implicit termination facilities and which relate effects obtained using the recursionless construct in conjunction with its control variable.

These effects are analogous to those obtained in factoring the loop from gotoless languages, and the top down operators are directly (vertically) analogous to the various forms of usage of accumulation and coapplication from the original basis. The side benefits of simulating mutual recursion and multiple argument recursive functions arise from these operations.

The only serious problem concerns "corecursion", insofar as termination of top down generation is not factorable in the same sense as is sequential cogeneration. More precisely, an implementation of:

```
s . fs1 . fs2
```

is able to pulse generators for "s", "fs1" and "fs2" in a loop, apply the functions and pulse all three again. The same is true for "!":

```
t ! fs1 ! fs2
```

We can recur to the first terminal node, pulsing at each recursion level both "fs1" and "fs2", applying the resultant function, etc. This is permissible because:

```
t ! fs1 ! fs2 ≡ t ! (fs1 .  fs2)
```

However, this association is not possible using "↓" or "∀", for

```
t ↓ fs1 ↓ fs2 is not ≡ t ↓ (fs1 .  fs2)
```

and hence, if "fs2" is a termination function (a tree trimmer) we cannot obtain the effect of the cosequential application. This is not the case with iterative accumulation--it can be executed cosequentially with coapplied or coaccumulated functions. The problem arises from the lack of a result definition which is defined "stage-wise" with recursion. For example, the function:

```
<<0>; 1; <2>> ↓ (. (rid *) . < <<0>; 1; <2>>; 1; <<0>; 1; <2>> >)          [15]
```

first defines

```
<<<0>; 1; <2>> %; 1; <<0>; 1; <2>> %>
```

and recursion will occur where the "%"s have been placed. The second level of recursion will generate:

```
< <<<0>; 1; <2>>%; 1; <<0>; 1; <2>>%>;
  1;
  <<<0>; 1; <2>>%; 1; <<0>; 1 <2>>%> >
```

Because there is no intermediate representation of these stages of recursion analogous to the stages in iterative accumulation, infinite generations cannot terminate in the same manner. (In essence, it is as though accumulation were defined as the **val** of its current definition; the accumulation sequence **per se** would then be inaccessible.)

Although we can deal with this problem in terms of coroutines (see Chapter IV), a development which makes the various stages of the recursion part of the result would be preferred. We simply do not see how to do this currently, but believe it can be done.

6. Bottom up

A second candidate for a "recursionless" construct is a "bottom up" operator. In developing a bottom up recursionless construct we proceed exactly as with the top down operator. A bottom up algorithm is frequently explained in terms of "reducing the handle" of a tree, in the jargon of translator writing formalists [FG]. In essence, a function is applied to and replaces the deepest nodes in a tree before it is applied to their superior nodes.

A recursive formulation of such an operation for a tree "t" may be expressed:

```
t f :: t tp then (t tf) else (t . ('f *) bu)                              [16]
```

A recursionless construct for such a form in an Algol-like language could be:

```
<recursionless construct> ::=
  bottomup <variable> ← <recursive structure>
  <termination specification>
  do <bottom up function body>.
```

Again the <variable> refers to the entire subnode within the body, predicate, and termination function, but the structure must be a tree (unlike top down operators, where any value which causes a sequence to be generated is acceptable). This requirement arises simply because the operand structure must initially contain terminal nodes; a bottom up algorithm cannot generate the structure to which it is applied; the top down algorithm must. For example, a bottom up operation which sums the elements at the terminal nodes of a binary tree, "t", might be written:

```
bottomup T←t until atom(T) whence T do T[1]+T[2].
```

In developing a bottom up operato- for inclusion in the basis, we proceed as with the top down case, by defaulting the termination predicate to "atom?" and introducing a sequence of bottom up operators or "sections". The bottom up operator, "↑", may be written:

```
t ↑ bu :: t atom? else (t . ('↑ (bu tail)*) (bu 1st))                    [17]
```

It retains the correspondence between function sequence index and tree node depth as before. However, the last element of the sequence (corresponding to the last terminal node) is applied first.

A recursive evaluation procedure which sums the elements at the odd depths of a binary tree, and takes their difference at the even depths, may be written:

```
s sum :: s 1st + (s 2nd);
s diff :: s 1st - (s 2nd);
```

```
t sumdif :: t ↑ (<sum; diff> gen *)
```

For example,

```
<1; <2; <<3;4>; 5>>> ↑ (<sum; diff> gen *)
 ≡ <1; <2; <<3;4>; 5>>> .  (↑ <diff; sum; diff...>*) sum
 ≡ <1 ↑ ...; <2; <<3;4>; 5>>↑<diff; sum; diff...>> sum
 ≡ <1; <2; <<3;4>; 5>> .  (↑ <sum; diff; sum...>*) diff> sum
 ≡ <1; <2 ↑ ...; <<3;4>; 5> ↑ <sum; diff; sum...> >diff>sum
 ≡ <1; <2; <<3;4>; 5> .  (↑ <diff; sum; diff...>*) sum>diff>sum
 ≡ <1; <2; <<3;4> ↑ <diff; sum; diff...>; 5 ↑ ...>sum> diff> sum
 ≡ <1; <2; <<3;4> .  (↑ <sum; diff; sum...> *) diff; 5> sum> diff> sum
 ≡ <1; <2; <<3 ↑ ...; 4 ↑ ...> diff; 5> sum> diff> sum
 ≡ <1; <2; <<3;4> diff; 5> sum> diff> sum                          [18]
 ≡ <1; <2; <3-4; 5> sum> diff> sum
 ≡ ...
 ≡ 1+ (2- ((3-4) + 5))
```

Although termination continues to be a problem, it is clearly a separable problem: both top down and bottom up have the identical <termination part> specification in the recursionless constructs. In particular, the "!" operator permits application of the termination function for both operators, separably.

A more general form of bottom up operation allows the recursive traversal of the argument tree and the subsequent possibility of retaining the original node as well as the bottom up value at each level of recursion. The recursive form below allows the bottom up function to be binary:

```
t f :: t tp then (t tf) else (t . ('f *) bu t)                    [19]
```

Notice that in the expression [18] the action of the bottom up function has meaning even if the operators "sum" and "diff" had been binary. In fact, this is permitted; the bottom up function is defined as [18] exemplifies, for binary functions. That is, the "tree" of "sections" defined by the bottom up form [17] is the value of the operator when the sequence of functions consists of binary functions. A section so defined will be referred to as a "recursive section".

To obtain the effect of [19], the bottom up function is extended to allow one of its arguments to be a recursive section (instead of a sequence of sections). The structure of the recursive section participates in the operation in the following way. The tree structure of the argument to the section must be identical to the other argument tree (or have a terminal node where the argument tree does not). The functions are then applied bottom up to the corresponding pairs of nodes.

For example,

```
<a; <b; c>> ↑ (f *) ↑ <q; <r; <s;t>>>                                [20]
  ≡ <a; <b;c>f>f ↑ <q; <r; <s; t>>>
  ≡ <a; <b;c> f <r; <s; t>>> f <q; <r; <s; t>>>
```

In one sense, the "recursive section" is the only truly recursive representation for a "program" we have dealt with. In particular, each of the recursive operators imposes a recursive interpretation on sequences of functions used in a recursive control context. The recursive function, by contrast, contains the recursive structure explicitly. Later in this chapter we deal with the significance of this structure more fully.

More general forms

The top down and bottom up operators mimic the standard notions which they represent. Not to belie their significance when used alone, it might appear as though a plethora of "recursionless constructs" are required to cover recursive functions in general. That is, one might feel significantly constrained were the recursioness operators used in lieu of recursion in the basis--much more so than with the gotoless constructs.

We do not feel this should be so, and divert our attention to recursion in general, momentarily, to substantiate our convictions. We are primarily interested in two questions:

1. When is a recursive algorithm preferrable to a sequential equivalent?
2. What aspects of recursion are not captured by the recursionless operators?

To approach the first question, we notice that it is not the case that all functions defined on recursive structures need themselves be recursive. For example, if a sequence "s" is a path in a tree (a sequence of successive indices of subnodes), the node at the end of the path in tree "t" is:

```
t / (sub * .  s) val
```

A recursive formulation is unnecessary:

```
x f p :: p isemptyseq then x
                        else (x sub (p 1st) 'f (p tail));
```

The point is simply that recursive data structures do not necessarily require recursive accessing functions.

The recursive implementations of "." and "/" [2] demonstrate that recursive control structures need not deal with recursive data structures. In Chapter II (p. 56) a recursive schema was presented which was also not "essentially recursive"; however, a brief study of that schema is in order. It may be reformulated without termination conditions in terms of the basis as follows:

```
x f :: x td 'f bu x                                        [21]
```

This function may be written iteratively as a double accumulation:

```
x f :: terminal-value/(bu*.(x/(td*) reverse))val           [22]
```

where "terminal-value" and the termination of the inner accumulation depend on the omitted predicate and terminal functions. In essence, the reverse of the top down accumulation is the "argument stack" sequence, which is then an argument to the bottom up accumulation sequence.

In effect, no control information is needed in the stack--that is, the return point position is fixed. We therefore say the function is not "essentially" recursive. Indeed, the recursive implementations of "." and "/" [2] are of this form, and the iterative definition is preferred. (Both would pulse the sequential arguments, but a recursive implementation wastes "stack space" by storing a constant return point.)

Only a slight modification to the schema is required to produce an "esentially" recursive function--one in which the bookkeeping of the return point is non-trivial and justifies a stack implementation. Consider a recursive function whose body contains several recursive calls:

```
x f :: ...'f ...  'f ...  'f.
```

In fact, this form of function is "essentially" recursive, for the context of the call (the bookkeeping of the return point) is non-trivial (it would be very difficult to do iteratively--in fact, it would require pulsing a data structure via "push" and "pop" primitives). These calls are either multiply recursive--i.e. f(g1(f(g2(x))))--, or they may be executed independently, dependent on context--i.e. bu(f(td1(x)), ... , f(tdn(x))).

A stronger generalization would permit the recursion to occur within a loop. This form would also be "essentially" recursive, for the bookkeeping of the loop indices at the various levels of recursion would be non-trivial.

Both cases (excluding multiply recursive calls) require that recursion occur in some sequence within the recursive function body. Thus, a more general recursive form--and, indeed, an essential one, may be derived (again without termination conditions):

```
x f :: x td . ('f * bu x)                                    [23]
```

Notice that, for td ≡id (the left identity function), x f is a combination of the bottom up forms and for bu ≡ lid (a left identity ignoring its right argument), x f is a top down accumulation form:

```
x id .  ('f * bu x) ≡ x ↑ (bu *) ↑ x;
x td .  ('f * lid x) ≡ x ↓ (td *)
```

But notice particularly,

```
x td . ('f * bu x) ≡ t ↑ (bu *) ↑ (x ↓ (td *))                [24]
```

where t is a function of "x↓(td*)" and the termination function--e.g. "x ↓(td*)!(tf*)". That is, the quite general recursive form [23] is equivalent to a separable application of the top down and bottom up forms.

The "double accumulation" analog between [22] and [24] is particularly striking. It is as though the top down form were an accumulaton which branches at each subnode (forks) and bottom up is a "merging" form of accumulation (joins). Both forms are "essentially" recursive, for a stack is required for the index of the loops in [12] and [13].

Actually, more general forms involving multiple arguments and mutually recursive functions are obtainable. In particular, if we introduce a new notation to allow multiple arguments (in excess of two) to operators, we can demonstrate the extreme complexity of functions which can be composed using the recursionless operators. The notation simply requires multiple arguments to be in brackets. Both

```
[a;b] f c :: body
```

and

```
  a f [b;c] :: body
```

represent three argument operators whose calling sequences are:

```
  [actual-1; actual-2] f actual-3
```

and

```
  actual-1 g [actual-2; actual-3],
```

respectively.

The function,

```
  t f [td; tf; bu] :: t ↓ td ! tf ↑ bu                                  [25]
```

is equivalent to the recursive implementation:

```
  t f [td, bu, tf] :: t rf [td 1st; tf; td tail; bu]                    [26]

  t rf [acc; tf; td; bu] ::
        (tf emptyseq else (td emptyseq) else (bu emptyseq))
        then t
        else ((t atom?) then (t (tf 1st)))
        else (t .  ('rf [t (td 1st) acc; tf tail; td tail; bu tail] *) (bu 1st))
```

This is indeed a fairly general recursive form, and the implementation of the function does not even include the effects of escape functions in the recursive programs!

The application of corecursive operators should not be implemented sequentially--that is, the top down operator should not be applied to the entire tree before proceding to apply the next top down operator, etc. In fact, the functions [26] should be the implementation for the function [25]. The "recursionless" operators are clearly "corecursive" in the same sense as functions of sequential objects using coapplication and accumulation are cosequential.

A Recursive "*"

Although the corecursive operators may be used to replace the recursive functions, it is not clear that the basis contains any "recursionless constructs" analogous to the "gotoless operators" of the initial basis. In particular, we can draw the following parallels between the sequential constructs and recursive constructs:

1. Constant representation: <1;2;3> sequence and <1;<3;5>;<9;<6>;7>> tree;

2. Cosequential/Corecursive: "." & "!" and "/", "∀" & "↓";

3. Gotoless/Recursionless: "*" and ?.

There is no obvious potentially infinite recursive form to correspond to the potentially infinite sequential form, the loop.

In fact, the corecursive operators impose a recursive interpretation on **sequences** of functions; the unbounded recursive elements to correspond with loops arise from using loops on funtions which (sequences) are then interpreted recursively by the corecursive operators.

The correspondence between the corecursive and cosequential constructs can be emphasized much more strongly if the loop is considered to be a form of "quote" operator. To understand such an interpretation of the loop, consider the expression

```
s . <1;2;3> . p
```

If it is desired to suppress the "normal" action of cosequencing the sequence "<1;2;3>", the loop operator is used, viz.

```
s . (<1;2;3>*) . p                                          [27]
```

"Quoting" is generally understood as the act of suppression of the normal interpretation, and hence, "*" may be viewed as a "quote" operator of sorts.

Now, if the corecursive operations were modified to "coapply" two **recursive** structures, instead of using the interpretation between sequences and trees imposed in the definitions above, the notion of a recursive "quote" in the same sense as for "*" arises. For example, in the bottom up operator discussion, it was convenient to define a "recursive section" consisting of a sequence of binary operators applied (bottom up) to a recursive data structure (see [19] and [21]). If the top down operations were defined analogously, a recursive quote "$" could be defined which imposed the recursive sequence structure used above. That is, to obtain the effect of:

```
    t ↑ (ev *)
```

we would have to write

```
    t ↑ (ev * $)
```

to suppress attempts to interpret "ev*" as a recursive form, just as "*" suppresses attempts to interpret its argument as a sequential form.

This presents a more unified view of corecursion and cosequentiality, for now we can consider the effect of recursive "data" structuring using the recursive quote, "$". Previously, the interpretation of the sequence of functions was accomplished by the corecursive operators themselves; i.e., recursive interpretations of sequences were confined to sequences of functions. With the recursive quote "$"--a true recursionless construct--sequences of "data" can be considered. That is, we can define a function on the recursive representation, relying less on whether it represents a program or data.

This notion represents the fringe of our understanding of the recursionless constructs' interactions with the language basis. Obviously, several different recursive quotes could be considered; this might be significantly more complex than imposing a recursive interpretation depth-wise on a sequence as has been done above. Although future considerations of recursionlessness and corecursion should probably be based in part on this recursive quote, the implicit (recursive) quote in the corecursive operations should not be disregarded. Even in the initial basis, it would be quite consistent to permit "*" to be imposed implicitly; for example, there is little reason not to permit the implicitly quoted interpretation:

```
    a . + . b ≡ a . (+ *) . b
```

for "+" simply does not have an interpretation as a sequence. Obviously, the explicit use of the quote must be permitted for cases such as [27] above. There is no ambiguity between:

```
    <f; f> and f<1;2>
```

with respect to how to treat them as operands of corecursive operations--the former requires the (implicit) "$", the latter does not. Hence, the formulations above of the corecursive operations may ultimately be preferred; however, the notion of a recursive "quote" should be considered for a future formulation of the basis.

Recursionlessness and structured programming

We emphasize that our considerations of recursionlessness are preliminary. Although we are able to express significantly complex recursive functions (see [25] and [26]), we do not feel we have more than "scratched the surface" of what might be interesting and useful.

In terms of **eliminating** recursion, we are not convinced that we have as strong a case against recursion as we do against the goto. Although the potential for misuse of recursion is at least as great as for the goto, the actual (observable) misuse is not. This arises from users' qualms over the inefficiency of using recursion at all in programs, and in general, from confinement of recursive programming to academia (at least in the U.S.). That is, if students were taught that recursion is as important and useful as the goto, the programming dilemma might be considerably more complex.

We are thus in a position of being able to structure a potentially dangerous concept before it actually becomes dangerous. Such structuring is useful in its own right--for example, it begins to eliminate the detail arising from implementing algorithms which are naturally expressed as "top down" or "bottom up".

Historically, would we have had to advocate eliminating the goto if news of its existence had awaited the last few lessons of instruction in programming courses?

CHAPTER IV

CODEPENDENT STRUCTURES

In Chapter III it was mentioned that some programs written in the initial language basis were unduly complex because of their lack of a "clean functional decomposition". This chapter examines the nature of such programs and ultimately shows that their concise specification hinges on the notion of coroutines or "codependent structures".

We shall be dealing with structures which are described independently and each of which can be thought of as being in some "state" at any given time. Such a group of structures will be referred to as "coroutines", although the traditional notion implies that the "state" include a program counter, which is not always necessary here. In this chapter, we are concerned with the extent to which the independently described structures can and should depend on the states of each other.

Coroutines in Applicative Languages

Although there are many examples of programs whose implementation is made more efficient through the use of coroutines†, it is somewhat more difficult to justify coroutine control from a structured programming point of view. We are not concerned with justifying the coroutine control present in the cosequencing and corecursive operators, for the decomposition there is essentially functional. However, if we move to a more general coroutine structure, issues involving global variables and side-effects emerge.

In one sense, the argument for the inclusion of a coroutine mechansim is a counter-argument to the primary argument for an applicative langauge. In an applicative language, identical expressions in the same static context have identical values; that is, functions are well-defined in an applicative language. This allows the programmer to depend on the preservation of relations on the environment over control constructs such as function calls.

- -

†Frequent reference has been made to compiler decompositions, for example.

From a structured programming point of view, programming in an applicative language may become dangerous because the programmer might become dependent on such invariant relations. For example, the invariant relations may arise from the way in which a data structure has been implemented, and assumptions based on these relations may then become an integral part of the program. Changing the data structure would then be impossible, even though the changes were consistent with the original specification of the problem. That is, we must distinguish between apparent structure and the implementation of that structure. In and of itself, this is actually only an argument for a data structuring mechanism. However, it will be shown that the explicit subordination of one structure to another required in an applicative language presents particular problems to modifying the program structure.

To understand how this arises, and in particular, to understand how a problem can lack a "clean functional decomposition", a LISP 1.0 program will be rather thoroughly dissected. LISP 1.0 is an applicatve language, and, hence, all problems require a functional decomposition whether it be "clean" or "unclean". The problem to be solved by the program is intentionally unrealistic: given two lists, "def" and "s", the function "de" below will produce a result whose elements are those of "s" except where elements of "s" are less than "3". In those cases, the elements chosen will be successively from the list "def". Thus,

```
s = (1 3 2 4);
def = (7 9)

de[def;s] = (7 3 9 4).
```

Two LISP functions which accomplish this are:

```
de[def;s] = [null[s] ⇒ NIL;                                        [1]
             lessp[3;car[s]] ⇒ cons[car[s]; de[def;cdr[s]]];
             T ⇒ F[def; s]];

F[def;s] = cons[car[def]; de[cdr[def];cdr[s]]];
```

("F" is separate for explication below; we assume "def" is of sufficient length that we cannot run out of default values.)

First notice that the body of "de" is free to reference "def" in any way desired. For example, there is no protection from using "caadr" on it. This argues for a means of structuring "def" in the sense of constraining its accessors to a particular set of functions--"car" and "cdr" in this case.

Notice also that "s" and "de" are "corecursive"--each time "de" is called, "s" is pulsed by taking its "cdr". This is not the case with "def"; however, "def" and "F" are corecursive in the same sense. Although we might immediately imagine that a decomposition which emphasizes this cosequentiality is possible, we first consider the implications of allowing the element of "def" chosen to depend functionally on the elements of "s" which are less than "3". This can be accomplished by respecifying "F" as:

```
    F[def;s] = cons[f[def;car[s]]; de[def;cdr[s]]].                    [2]
```

where "f" accomplishes the functional dependency. If we were able to instantiate "def" with "f" previous to the execution of "de", or were able to define it global to the functions called by "de", there would be no need to pass "def" as a parameter. The implementation below could be used.

```
    de[def;s] = de'[s];                                                [3]

    de'[s] = [null[s] ⇒ NIL;
              lessp[3; car[s]] ⇒ cons[car[s]; de[cdr[s]]];
              T ⇒ F[s]];

    F[s] = cons [f'[car[s]]; de[cdr[s]]];

    f'[e] = f[def; e].
```

This implementation uses the LISP binding which permits "def" to be global to all the functions called by "de". This is subject to the same dangers as [1]--i.e., there is no way to confine the access of "def" to the call "f".

Now, what if "f" were to be programatically dependent on its calling sequence--that is, what if "f[def;s]" were different dependent on the number of times it has been called from "F"? A particular example for "f" will help to illustrate the problem. Notice that [2] or [3] could not be used to implement [1], in which the element of "def" selected depended on how many had previously been used. Hence, if we desire "f" to act exactly as [1], but additionally insist that it add the element of "s" to the element of "def", the need for a new programming device arises.

What is frequently used in such a situation is an encoding device--the value of "f" must be encoded with the updated state of the computation which we desire for "f". In this case, the updated state will be the "cdr[def]". Thus, we can accomplish the effect by defining F (in scheme [1]) as:

```
F[def;s] = cons [car[f[def;car[s]]]];                              [4]
                 de[cdr[f[def;s]];cdr[s]]

f[def;e] = cons [add[car[def];e]; cdr[def]].
```

This decomposition is what is meant by an "unclean functional decomposition" to a problem. In essence, we have a program "f" which produces a sequence of values depending on "def" and an argument sequence of "e"s. However, the program "f" is considered entirely subordinate to "de", and its state must be continually passed around as a parameter.

If this subordinance were important from a structured programming point of view, then the above functional decomposition ([4] with [1]) is to be preferred. However, from the statement of the problem, there is no reason to prefer implementation [4] to one like [3]. To be precise, even if "f" had the side effect that the "cdr[def]" replaced "def"--which would enable us to define [4] more concisely as [3]--there would be no effect on the relationships to be considered in "de".

The only objection to [3] is that the **global** variable "def" is accessible by "de". If there is a way to specify the existence of two programs--each with its own state variables--and limit their references to each other to a functional interface, this objection is removed. This is almost an exact definition of "module" according to Parnas [PA], and the theory surrounding his work has a definite bearing on the coroutine facilities about to be introduced into the basis. It is also consistent with the efforts toward constraining **global** variable usage [WS].

It is particularly interesting to note that the notion of side-effect--which the above implementation introduces--can be independent of "assignment". That is, no notion of assignment ever enters the basis--yet codependent structures introduce the notion of side-effect.

Our approach to the inclusion of coroutines in the basis is as follows. First, the coroutine nature of the cosequencing operators is examined, and operators are developed to introduce more general coroutine facilities, which are presented next. Finally, some implications of coroutines to data structuring are discussed, followed by a brief discussion of the implementation of the basis in terms of coroutines.

As in Chapter III, it should be emphasized that the operators presented in this chapter are tentative. We are more interested in explaining the desired effects than in proposing a concrete syntax for their specification. Also, the reader should be forewarned that the operators introduced in this chapter do not produce expressions which are significantly more concise than corresponding Algol programs, for example. This effect arises from the more primitive nature of the coroutine operators themselves; effectively, they introduce the ability to explicitly pulse sequential structures. Much has been made of the lack of such a requirement in the basis to this point; here we examine where the ability to pulse structures is desirable, if only to define and study higher-level cosequencing and corecursive operators.

Some Remarks on Names

A brief digression is now in order. The basis operations generally tend not to impose special interpretations on names, but the facilities presented in this chapter rely on names to a much greater extent. In particular, none of the operators in the basis at this point is defined with a name as a required argument, nor does any defined operator give a preferred interpretation to a particular name as, for example, the **for** construct in Algol gives preferential treatment to the control variable.

This lack of reliance on names for semantics is intentional. Any construct requiring a name increases the number of names temporarily introduced by the programmer--a phenomenon the "operator" notion avoids (see Chapter 1). Also, scope issues are frequently very complex, and when semantics can be specified without their involvement, a description is often simplified immensely. We advise this language design technique: defer issues of names as long as possible.

We do not, however, deny the language enhancement that names can and do provide. In particular, the escape operators should be extended to allow named control context escapes, and such are included in the final basis [WU,1972]. Also, where temporary functions are required--as occurs frequently with accumulations--scope control such as block structure should be permitted to localize the definitions and possibly even control variables or named accumulators. However, the lack of such facilities has been a very effective aid to simplifying the presentation of the basis to this point.

In the operators presented below, names cannot be ignored as easily--in fact, some of the operators would be needlessly complex without reliance on names. This does not reflect a change in philosophy, but rather a concession to the more primitive nature of the coroutine operators.

Cosequencing Reexamined: Partial Cosequentiality

By way of introduction to coroutines, the cosequencing operators "." and "/" are reexamined. In particular, notice that in the expression "data . program", the data is cosequential with the program. Also, the result is cosequential with both the program and the data. That is, not only does "." identify data element "i" with program element "i", but also the value of "." is a sequence whose "ith" element can be identified with the "ith" element of the data or program sequence. There are programs in which one such cosequential identification can be made but not the other two (i.e. program with data, program with result or data with result). They are considered separately below:

1. Non-cosequential result: the emit operator

An example in which the program is cosequential with the data but not with the result is the "mask" operation; the nonempty elements of the argument sequence make up the result sequence of the "mask" operation (see Chapter II or Appendix III). If the implementation does not rely on "<> **gen**" (see discussion, Chapter II), the function must be written using an accumulation:

```
    s tf x :: x then (s conc <x>) else s;                    [5]
    s mask :: <> / (tf * .  s) val;
```

In this implementation of the mask function, no element of the masked sequence can be produced until the entire argument sequence (s) has been generated (by virtue of the **val** operation). Unless the implementation is exceedingly clever and notices that the result sequence only changes by appending, an unbounded argument cannot be used. Even then, the semantics of **val** should ensure that the sequence terminates; i.e., the expression "1 * **val**" should be undefined.

However, it is clear that in a simple scan across the sequence an element could be output (entered into the result sequence of "mask") whenever it is nonempty. The **emit** operator is defined to accomplish this, and actually constitutes an ability to "pulse" the output sequence, or to explicitly "generate" elements one at a time. The operator outputs its (left) argument as an element of the result sequence for the innermost sequential expression in which it is embedded. The mask function above [5] can then be rewritten:

```
    x thenf f :: x then (x f);                               [6]
    s mask :: s .  (thenf emit *);
```

Then,

```
    <1; nil; 3> mask
      ≡ the sequence emitted from
        <1 then (1 emit); nil then (nil emit); 3 then (3 emit)>
      ≡ <1; 3>
```

Only emitted values constitute the result of a sequence expression containing an **emit**. The expression value of the **emit** is its argument.

The introduction of an emit operator which requires a name as operand is particularly useful, and motivates the second operator of this section. The named emit operator--**emitn**--hinges on the notion of an "emittor-collector" expression, which is of the form†:

emittor-expression : [collector-name$_1$; collector-name$_2$; ...]

The **emitn** operator requires a collector name as its right operand, and simply emits its left operand to the named collector. (Its value in the expression, as with **emit**, is its left operand.)

A collector is simply a named entity which accumulates the elements emitted to it in a sequence. The emmitor-collector expression (hereafter abbreviated "EC") defines a result which is a set of named sequences. An element of the collected sequence set may be selected by name as though the name were an operator; e.g.,

exp : [a;b] a ≡ sequence emitted to "a".

This may be clarified if one thinks of a set of associations specified:

[name$_1$ :: value$_1$; name$_2$ :: value$_2$; ...].

Specifying any name after the set selects the value associated with that name in the set: viz.

[a :: 1; b :: 2] a ≡ 1

- -

†Actually this is a simplified version of the emittor-collector expression. It will be embellished throughout this chapter.

Thus, the emittor-collector expression potentially has a set as its value.

The following example defines an operator "eo" which produces an EC whose collectors have the even- and odd-indexed elements of a sequence "s" as collectors named "even" and "odd":

```
s eo :: s. (<emitn odd;emitn even> gen *) : [even; odd];            [7]

<1;1;2;3> eo odd
  ≡ sequence emitted to "odd" in
     <1·1;2;3> .  <emitn odd; emitn even; emitn odd; ...> : [even; odd]
  ≡ <1; 2>
```

The EC expression is especially useful when several functions share a common complex control structure. It also represents the beginning of an association mechanism which is of significance to the concept of "structured data", discussed later in this chapter. The collector portion of ECs is generalized presently.

2. Non-cosequential operand: the collect operator

The **emit** operator was introduced to permit program and data cosequentiality, without requiring either to be cosequential with the result. The **collect** operator, to be introduced presently, facilitates writing programs which have cosequential program and result, but not cosequentiality of data and program or data and result. The accumulate operator is already of this form: only the program and result are cosequential. Frequently, functions are written which "pulse" an input sequence--the accumulated argument. This pulsing is in terms of "1st" and "tail" in much the same way as LISP functions use "car" and "cdr" (see Chapter III and [1]-[4] above). Here the ability to explicitly pulse a codependent sequence is introduced.

The problem used to explicate the "unclean decomposition" (see [1]) is now reused to introduce the semantics of the **emit** operator. In particular assume that a sequence called "def" is to be used to replace elements which are smaller than "3" in a sequence "s" by a function named "de". This may be written in the initial basis:

```
i f a :: <a then i else (i+1);
          a else ('def sub i)>;

def de s :: <1; nil> / (1st f * . (s . (ge 3 *))) . (2nd*)           [8]
```

The reader need not understand the particulars of this function, but in essence, the accumulation keeps track of the index of the defaults sequence to be used in the event of an empty element in s. Thus,

```
<100; 200> def <3; 1; 4; 2>
```

would produce the accumulation:

```
<<1;3>; <2;<100;200> sub 1>; <2;4>; <3; <100;200> sub 2>>.
```

The selector ". 2nd *" then produces the result: <3; 100; 4; 200>. In effect, the accumulation "pulses" the default sequence (by incrementing the index used for selection). The **collect** operator permits the ability to pulse an argument sequence directly.

The **collect** operator is introduced as an extension to the EC expression above. Instead of a set of collector names, a simple collector expression is permitted, in conjunction with an emittor expression which uses the emit operator (and not the **emitn** operator). In addition, any sequence may be used to stand for an emittor which emits that sequence. For example, the function "de" [8] may be rewritten:

def de s :: def : (s . (**ge** 3 **else collect** *)) [9]

The value of the **collect** operator is the element pulsed from the emittor. The value of an EC of this form is the sequence to the right of the ":". Thus,

<100; 200> def <3; 1; 4; 2>

≡ <100; 200> : (<3; 1; 4; 2> . (**ge** 3 **else collect** *))
≡ <100; 200> : <3; **collect**; 4; **collect**>
≡ <3; 100; 4; 200>.

Once again, the introduction of an interpretation reliant on names is useful. The emittor expression is now permitted to be a set of named emittors; again none of the emittors may reference named collectors. Such a set is specified:

```
[emittor-name1 :: emittor-expression;
 emittor-name2 :: emittor-expression; ...]
```

Then the **collectn** operator is introduced to selectively "pulse" the emittors by name. Its value is the pulsed element.

To illustrate, the following EC interleaves the elements of two sequences into one:

```
s interleave p :: [s::s; p::p] : (<collect s; collect p> gen *)
```

Assuming a collector expression terminates if it attempts to collect a terminated emittor, the above function will produce:

```
<1;2> interleave <3;4>                                          [10]

   ≡[s::<1;2>; p::<3;4>] : <collectn s; collectn p;; ...>†
   ≡ <1;3;2;4>.
```

It is important to note that the collector expression is controlling the pulsing of the emittors, and not vice versa. That is, this construct should not be confused with that of several languages which permit a loop driven by an emittor (generator) [SC]. (The initial basis permits this latter facility quite easily.)

To summarize, the **emit** operator has been introduced in order to permit cosequentiality of program with data, without insisting on the cosequentiality of either with the result. This capability corresponds directly to the notion of a generator or "pulsed" output. Similarly, the **collect** facility was introduced in order to permit cosequentiality of result with program, but not insist on the cosequentiality of either with the data. Analogously, this facility provides "pulsed" input from a generator.

We are about to proceed to a discussion of a more general coroutine facility. It will be useful to have the emittor/collector expressions summarized syntactically in BNF††:

```
<emittor-collector> ::=<simple emittor-collector>
    /<join emittor-collector>
    /<fork emittor-collector>

<simple emittor-collector> ::= <unnamed-collector emittor-expression> :
    <unnamed-emittor collector-expression>                      [11]
```

---

†A quote problem becomes quite pronounced here: do the **collects** occur before the looped expansion or not? For now, assume not.

††The notation <x>-list is used to indicate a list of <x>s separated by semicolons.

```
<join emittor-collector> ::= <emittor-set> : <named-emittor collector-expression>

<fork emittor-collector> ::= <named-collector emittor-expression> : <collector-set>

<emittor-set> ::= [<emittor association>-list]

<emittor association> ::= <name> :: <unnamed-collector emittor-expression>

<collector set> ::= [<name>-list]
```

Where the "unnamed" expressions (e.g. <unnamed-collector emittor-expression>) contain neither **collectn** or **emitn** (in <unnamed-emittor...> and <unnamed-collector> respectively), and the "named" expressions contain only those constructs. To reiterate, any expression producing a sequence may be used as an <unnamed-collector emittor-cxpression> and the value of an EC is either a set of named sequences or the collector expression. Of course, ECs are simply expressions, and may be used in any context where an expression may be used.

Codependency

The **emit** and **collect** operations quite clearly require a coroutine structure for their implementation. In this section the relationship of these operations to a more general coroutine facility is discussed. It will be shown that the operations are not fully adequate for expressing programs which do not admit a clean functional decomposition; in particular, the nature of "codependent" structures is examined more closely.

The **emit** and **collect** facilities **do** permit a cleaner decomposition than does the coresponding expression in the initial basis; they always eliminate an extra accumulated variable†. For example, compare [4] and [5] or [8] and [9]. This results from the factoring of the state of the non-cosequential sequence from the primary sequence (result from program and data in **emit** and data from program and result in **collect**). Although definitions of "coroutine" vary, they share the separation of states of processes as one aspect of coroutine control. It must be emphasized that coroutine execution is a sequential process; although the states of coroutines are separate they depend functionally on the sequence in which they invoke each other. In this sense, the **emit** and **collect** facilities define "codependent" expressions.

- -

†In the initial basis, a sequence is frequently used as the accumulated value in an accumulation. This sequence is often of fixed length, and the various elements of it are selected, much as variables in a program. See [5] and [8].

The distinction between a function call and a call to collect a value is extremely important. A function will be a constant; the **collected** value will not (in general). Neither codependent function can change the other's value; they can merely cause each to "change" their own value, by producing another sequence element.

The distinction between a fully general coroutine facility and the **collect/emit** facility lies in the extent to which the collected sequence can depend functionally on its calling sequence. A general coroutine facility permits this dependence to be parameteric. To explain such functional codependence the Bliss coroutine mechanism is examined briefly.†

In Bliss the ability to create coroutines (named "A" and "B" here) is provided. The precise syntax and mechanism used for the creation is not relevant. In essence, the ability to associate a control/data space (stack and program counter) with the named coroutines is provided. Assume that control resides in "A". Then a coroutine call, "exchange jump" (abbreviated **exch**), consists of an argument and the name "B":

```
    arg exch B                                                    [12]
```

much as the **emit** operation is used above. However, the value of the expression [12] is not "arg" as with **emitn**, but rather an argument to the exchange (in "B") which causes the return to "A".

For example,

```
    coroutine A(a1) = begin local t; t←(a1+2) exch B end          [13]

    coroutine B(b2) = begin local p; p←b1; p←(b1+1) exch A end
```

abstracts the Bliss facility for coroutine declarations. The parameters "a1" and "b1" are the initial parameters to the coroutines--the parameter of the first function call or exchange jump. They are undefined after the first exchange jump from within the body. Assume the coroutine can be invoked by the body of the block in which these coroutines are defined. Then the call "A(5)" will cause "A" to begin execution, with "5" as the value of "a1". "A" will immediately exchange jump to "B", as though a call of

- -

†In Bliss, the coroutine facility was introduced as an effect difficult to obtain without using the goto. The synopsis here is actually a modification of a much more general facility than is presented. Liberties have been taken with the syntax as well [WU,1970,1972].

"B(5+2)" were made. Control begins in "B" , with "b1" equal to "7". After it is stored in "p", the exchange is made back to "A", with the parameter "7+1". Execution resumes in "A" at the point of exchange to "B", with the value of the exchange expression as "8". Control then returns from "A" to the calling program. The final state of the local variables is:

t = 8; p = 7.

If the original call had been "B(5)" (instead of "A(5)") the state of the variables would have been:

p = 8, t = undefined.

Control would never return to the exchange in "A" in this case.

The point is simply that the coroutines depend functionally on each other--there is no input/output identification to be made. "A" appears as a function to "B", and "B" likewise to "A". Each coroutine presumes its task is primary and the coroutines it calls are auxilliary to it.

Examples of the utility of such a conceptualization are most often complex, for at least two non-trivial tasks must be dependent on each other, yet of distinct utility when standing alone. However, a conversational (interactive) language provides a nice environment in which such a conceptualization is enlightening. Consider two interactive chess-playing programs "W" and "B". A user with two terminals could play the programs against one another by allowing "W" the first move. He could enter "W"'s response as his first move to "B". "B"'s response could then be entered to "W", etc.

The user should feel quite trivial--he is acting precisely as an exchange jump†. Each program presumes the other to be its input function. Thus, if the two programs had been written with exchange jumps, and adequate naming facilities were available to make this dynamic connection [KR], the programs would have been "more general" in the sense that this frequently interesting activity was made easier for the programmer. Ths also establishes the activity of the human player (with either program) as a coroutine in nature. The implications to conversational system design are beyond the scope of this work; minimally, facilities to separate the user's "state" from the executing

†Actually the record of the moves is typed out to the user; hence, an intermediate coroutine would be required if anyone were interested in the progress of the game.

coroutines are necessary.

Coroutines can be introduced into the basis by extending the emitter-collector expressions to allow emitters to collect and collectors to emit. Taking the simple case first, <simple emittor-collector> in [11], each expression may contain unnamed **emits** and collects. For example,

```
x exch :: <x emit; collect> val;                                  [14]
x f :: x gt 2 else (x exch);
defaults deff s :: defaults .  (sub collect emit *) : (s .  f *);
```

defines a function "deff" which produces the sequence "s", (a subset of the positive integers), except where the elements of "s" are less than or equal to "2", elements from an array "defaults" are chosen. The elements from "s" which are chosen depend functionally on the value of the element in "s", and the number of defaults chosen to that point. (See [4].)

In particular,

```
<<100;101>;<200;201>> deff <3;1;4;2>                              [15]

   ≡ <<100;101> sub collect emit; <200;201> sub collect emit> :
       <3;< 1 emit; collect> val; 4; <2 emit; collect> val >
```

Assume control begins at the collector in the EC--i.e. to the right of the ":". Then "1" will be emitted to the left expression. This causes the left expression to begin evaluation and when the first **collect** is reached, its value will be "1". Using "%" for "program counters" or sequencer positions, the evaluation state at this point is:

```
<<100;101> sub 1 # emit; <200;201> sub collect emit> :
      <3; <1 %; collect> val; 4; <2 emit; collect> val>
```

The "#" is the program (control) sequence position. Evaluation proceeds, producing "100 **emit**" as the first element of the emittor expression. Control now changes back to the collector expression with an emitted value of the next **collect** in the collector expression. The evaluation of the collector side proceeds until its next **emit**, at which time the state of the computation will be:

```
<100 %; <200;201> sub collect emit> :
      <3; 100; 4; <2 emit; # collect> val>
```

Control resumes in the emittor expresssion until the **collect** is done, and the result "201" is computed (<200;201> **sub** 2 = 201):

```
<100; 201 # emit> : <3; 100; 4; <2; 7 collect> val>
```

The emittor's emit then causes control to resume in the collector, the value emitted by the collector is "201", and the sequence terminates. The value of the EC is the collected expression, and, hence, the value is :

```
<3;100;4;201>.
```

The simple coroutine expression above does permit the definition of functionally codependent structures. Although the emittor-collector relationship prevails--i.e. the collector is the value of the expresion--the subordinance of the emittor is not evident from an examination of the emittor-expression standing alone. For example, if the emittor and collector are interchanged in "def." [15], the value of the collector (the previous emittor) is "<100;201>".

If both sequences were of potential interest, both expressions would have to be specified ([15] and [15] with emittor and collector interchanged). In order to permit the use of both sequences without such recomputation the <fork emitter-collector> of [11] is extended. The <collector set> is expanded to allow a list of named coroutine expressions (which use unnamed **collect** and **emit** operators). The "<named-collector emittor-expression" may both **collectn** and **emitn** to the named coroutine expressions, and the result of such an expression is the association set of named coroutine expressions.

For example,

```
q ::  <collectn a emitn b; collectn b emitn a> * :                  [16]
  [a :: <3; 1 exch; 4; 2 exch>;
   b:: <<100;101> sub collect emit; <200;201> sub collect emit>]
```

has the value:

```
[a :: <3;100;4;201>; b :: <100;201>]
```

and the selectors "a" and "b" may be used functionally:

```
q a ≡ <3;100;4;201>;
q b ≡ <101; 201>.
```

In [16], the reader will recognize the expressions involved in the explications of [15] above. A description of the evaluation process analogous to that provided for [15] is presented in Appendix VI.

To complete the coroutine facilities, the <join emittor-collector> is extended to permit a coroutine set as the emittor portion of the expression. Again the elements of the emittor set expressions must use (unnamed) **emits**; we extend the facility by permitting (unnamed) **collects** in those expressions. This permits the controlling mechanism--the collector--to produce the value. For example, if the collector and emitter are interchanged in [16], the value of the expression will be the sequence of pairs:

<<1; 100>; <2;201>>

This introduces a flexibility into the language which is relevant to the works of Krutar [KR] and Parnas [PA]†. The ability to define named entities--coroutines, emittors and collectors--permits a dynamic linkage similar to that proposed in Krutar's work. Such a flexibility is consistent with the work of Parnas, but is not quite as general a facility as we presume he would desire.

To return to the introductory example of this chapter and the nature of "unclean functional decomposition", notice in particular how difficult the effect of [16] would be to obtain in an applicative language. To obtain the "defaults" subsequence, "<100;201>", the entire function would have to be rewritten. In the basis, we merely modify the order of the coroutines, or put them into "sets".

Before proceeding to a discussion of "data structures" and their relationships to coroutines, some mention of the lack of an explicit **exch** operator as the unique coroutine mechansim is warranted. Needless to say, the proposed mechanism is more general in that **exch** can be implemented in terms of **collect** and **emit**. Our reluctance to base the mechanism on **exch** involves initialization problems (parameters "a1" and "b1" in [13]). Our scheme allows the definition of coroutines which are of the nature of emittors--by using (...**emit**...**collect**...)--or of the nature of collectors--by using (...**collect**...**emit**...). That is, the former is able to **emit** (once) independent of any collected data, the latter is not. (Naturally, by using conditional facilities, more complex expressions can be built which are not so easily classified.) We do not have enough

- -

†These works are not easily related; however, they are both concerned with the ability to replace "modules" easily. This is the sense of relevance intended.

experience with the facilities to propose either as strictly preferrable from a language design viewpoint.

Data Structuring via Coroutines

The relationship of our concept of "data structures" to coroutines is extreme. To understand how the relationship arises, consider "conventional" data structures, such as arrays, lists, cyclic lists, stacks, etc. Each imposes a set of relationships onto the elements of the structure. The structure essentially translates into a set of accessing functions for items so structured, which can be used by the program (see [WU,1971] and [WG]).

The design of some data structures is such that a particular set of accessing sequences is assumed. In particular, a stack implementation enforces that the length of the sequence of pops done to the stack never exceeds the length of the sequence of pushes. The implementation of a FORTRAN array assumes that the accessing sequence is sufficiently random to warrant such a general structure (or that the combined effects of the structure's accessing sequences is best implemented with such a general structure). The implementation of lists presumes access will be to successive elements.

If we move to more modern data structures such as sparse arrays, paged arrays, files described as data structures, etc., such assumptions become even more pronounced. In fact, the nature of accessors for such structures requires a specification of how the accessor is being used. For example, in a sparse array "A", distinct accessors must be used in the expressions:

$A[3,4,5] < 0$ and $A[3,4,5] \leftarrow 234$.

Thus the accessing sequence is important to a data structure representation.

Going even farther, intended accessing sequences for the data structures arising in very complex programs such as operating systems, compilers and interpreters, become even more apparent from their conceptual (pictoral, verbal presentation) description. However, their description in terms of their implementation becomes complicated because of the inability to map different accessing sequences onto a group of elements to form a structure Instead, more primitive successor relationships must be imposed. In particular, either the relationships are imposed by a primitive pointer structure, or they are specified to a limited extent as a heirarchical entity and the program imposes the relationships whose specification is precluded by the enforcer of this hierarchy--be it a type mechanism or an applicative language.

The coroutine facilities are the beginnings of a structure mechanism which admits multiple (possibly complex) mappings onto a data structure. For example, a doubly-linked list may be used because it is desirable to sequence through the list in either direction. This may be expressed:

```
s prefix y :: <y; s gen>;

s dblylnk :: s .  (emitn forward emitn reverse *) :
                    [forward†; reverse::<>/(prefix collect *) val];
```

We may then define:

```
DIR :: <N; E; S; W; N> dblylnk
```

Then reference to "DIR reverse" will cause the creation of the reversed sequence. Naturally, a compiler is free to determine the implementation of such a structure, which might be a vector in this case, but would differ drastically if "dblylnk"'s argument is a magnetic tape file.

In the same vein, the coroutine primitives can be used to study and express structures which are modified by insertion, deletion, and assignment. Knowledge of the use of functions such as insert and delete not only affects the implementation of a data structure, but our conceptualization of the structure as an array, string, list, etc. The major reason the coroutine primitives are helpful in this area is that in the initial basis such considerations may be expressed in terms of **gen**††, which may in turn be implemented using **emit**:

```
s conc q :: <s gen; q gen>
   ≡ <s .  (emitn L *); q .  (emitn L *)> : [L] L
```

- -

†In light of the expanded emmitor-collector notation, this is an abbreviation for "forward :: **collect** *".

††Actually, **gen** is not an essential function in the following sense: for a potentially unbounded sequence "s", a function "s genf i" can be defined which produces "s" with its "ith" element **gened**, which does not depend on **gen** itself. See Appendix VIII.

```
s gen * ≡ s .  (emitn L *) * : [L] L
```

where "L" is a unique label.

For example, "insert", "delete", and "assign" may be expressed (in terms of a fixed sequence, "s"):

```
i assign y :: i-1 head s conc <y> conc (i rest s);
i insert y :: i-1 head s conc <y> conc (i-1 rest s);
i delete   :: i-1 head s conc (i rest s);
```

where

```
i rest s :: s .  (i head (id *) conc (emitn L *)): [L] L
```

We do not propose including these functions as primitives in the language. The basis to this point has demonstrated the extent to which we do not need assignment. It is much more important that the uses of insert and delete be approached in terms of the more general effect which they are used to accomplish. That is, we are not able to categorize the need for assignment yet; considerably more work is required in the direction of determining where we do not need it.

Orthogonal Issues

As we described in Chapter I, the language basis we developed was a priori constrained to attempting to describe the nested-sequential representation subspace of interesting programming structures. Although the remainder of the (semantic representation) space may be best described as the "nested parallel" space--with orthogonal elements of sets, association mechanisms, parallel operations, type mechanisms, name spaces, etc.--it would be inaccurate to say that nested sequential structures can be best described without the use of elements from this orthogonal space. For example, the selection gotoless construct **case** is semanticaly a parallel structure: retrieving an association from a set. However, it is very desirable to use this construct to express sequential program elements.

In the basis (as in LISP) this construct must be simulated by associating numbers with the elements of the set. Furthermore, the associated numbers are constrained to an initial sequence of the positive integers. Then we are able to count to the appropriate element by running through the sequence. However, it is actually semantically important that the distinction between a set and a sequence be delineated in future studies of the basis. (The specification that a program or data is a sequence may indicate that selection of an element from the sequence is not germane.)

Although the parallel space is conceptually orthogonal to the sequential space, many direct analogies from the language basis apply--even within the corecursive operators. Several examples may help to illustrate the point. In a discussion in Chapter II ("cosequencing operators"), the distinction of the coapply operator as a sequential operator was stressed, for the sequence could terminate at any point and could be unbounded. However, if the termination characteristics of the arguments to coapply are known in advance, the coapplication can occur in parallel. In fact, the compilation considerations of Appendix IV a.e appropriate in this domain, for we may even determine rather complex functions which can be applied in parallel using the same technique.

At a more primitive level, the emit and collect operators discussed above are quite similar to operations which spawn processes and wait for processes, respectively. In fact, any time an **emit** is encountered, the computation can "fork" (until a **collect** is encountered); any time the collect operator is encountered, the computation can "join".

Of course, the combined effects of parallel, sequential and nested representations is more complex than any in isolation. For example, sequences may conceptually change to sets for a parallel operation and back to sequential for output. More complex effects like the ability to map a sequence of names onto a sequence of values to produce a set of associations obviously parallel the semantics of coapplication, but are currently outside its domain. The recursionless constructs are potentially parallel each time a sequence of recursions must occur--i.e. everything described using the coapplication operator has a potentially parallel implementation given the proper constraints on the sequences. Thus, in effect the basis even at this stage is amenable to parallel **implementation** considerations; however, it lacks the means to express explicitly parallel effects. This is a deficiency, for the knowledge of parallel vs. sequential implementation drastically affects the algorithm chosen (parallel versions of good sequential algorithms may be less efficient than parallel versions of inefficient sequential algorithms).

The basis has been pushed to the point where orthogonal aspects (parallelism) should begin to be considered. For example, the recursionless constructs will be aided significantly by a type mechanism (for implicit termination) and an association mechanism. The lack of even a simple association mechanism for accumulated elements, for example, will probably thwart compilation efforts (or at least misdirect them) to some extent (see Appendix V for an example of how the lack of such a mechanism affects program conciseness adversely). Many attempts to program in the basis are made more difficult than is necessary because of the lack of even simple association and type mechanisms; however, defining them would have clouded the issues we wished to emphasize and would necessarily have been incomplete.

We emphasize: leaving out even simple orthogonal basis elements is an effective means for focussing on the issues at hand. We recommend this approach to language design.

Implementation issues

One of the primary reasons the coroutine primitives have been introduced is to move one level closer to an implementation. In particular, if one is able to refer to the most recently emitted value as "lastn", we can program both "." and "/" in terms of the coroutine primitives:

```
s .  q :: [a1 :: s; a2 :: q] : (collectn a1 (collectn a2) *);
v / s :: [fs :: s] : <v (collectn fs) emitn acc;
                    lastn acc (collectn fs) emitn acc *> : [acc] acc
```

Notice, then that the emittor set is effectively a declaration of new instances of the generator for the sequence to the right of the "::"s. This can normally be implemented very trivially in terms of a set of variables, a "program counter" and a pointer to the generating expression.

The above considerations might lead to an interpretive implementation. However, such an implementation is not necessary; some compilation considerations are given in Appendix IV. Although the details of those considerations are not important here, the fact that "." and "/" are operators defined in the language is important. In particular, an extensible language definition might prefer a "kernel" definition, which has the coroutine primitives as primitive and from which one may build "." and "/". The compilation considerations of Appendix IV are based on compiler knowledge of these particular operators and their relationships. That is, by defining a language "basis", we define not only the primitive operators but some of the operators which can be extended from a kernel, but which will be of obvious utility both conceptually and in implementation considerations. To define a language from the basis, the same step should be taken: more operators must be defined in terms of the coroutine primitives and "." and "/" (and recursion and "corecursive operators"). Ultimately, we may be able to eliminate the primitive coroutine facilities and provide the most useful effects of coroutines, using "coroutineless" operators.

CHAPTER V

CONCLUSIONS AND FUTURE DIRECTIONS

Several new ideas have been presented in this dissertation; each has caused the coining of a new phrase such as "recursionless construct", "cosequencing operator", "factored termination", or "implicit generation". In this chapter, we examine the extent of innovation represented by these ideas. We then present an overview of the extent to which the basis reorients our concept of control and data structures. Next the limitations of the language basis are considered, followed by some directions for future research along these lines. Finally, we consider the basis in the context of the order of magnitude criterion, discussed in Chapter I.

Innovation

Below we review the new ideas independently, then discuss their combined effects:

1. A pointerless representation

Explicit pointers in control or data structures are difficult to deal with in every approach to programming, principally because they vastly expand the relationships within and between data and control structures. That the gotoless constructs eliminate pointers from control structures suggested their potential utility for eliminating pointers from data structures as well. Although **analogies** between data structures and gotoless control structures have been drawn previously [HO,1968], the new idea in our work is to apply the gotoless constructs to a particular representation (nested sequences) independent of the elements of the representation. We then allow elements to be either programs or data. The most obvious benefit of this approach is to allow the explicit sequential representation of data structures without requiring explicit pointers to represent nested elements and cycles in the structures.

2. Operators relating data and control

Although LISP 1.0 allows the interpretation of pairs as sequences (lists), programs must explicitly "pulse" lists by using the "car" and "cdr" functions to impose a sequential interpretation. On the other hand, APL operators operate on structured data without explicit reference to the elements of the operand structures by imposing an element-by-element correspondence between operands. Such operators are defined to act in parallel on the elements of their operands;

we sought sequential analogies that emphasized the extent to which an element-by-element identification between sequential operands could be made. The "cosequencing" operators relate sequences in such a fashion, again independent of the elements of the representation. Thus, they tend to emphasize the extent to which data sequences follow the structure of the programs that use them, or equivalently, the extent to which control structures follow the data structures on which they operate. In essence, by providing a common interpreter for all structures in the language basis, the necessity to "pulse" data structures is lessened and operators which incorporate necessarily sequential effects are definable.

3. The partially instantiated function ("section")

The ability to describe cosequential activity hinges on the partially instantiated function--a function with only part of its argument list specified. The "section" generalizes such diverse programming objects as machine language instructions, Bliss data **structures**, and Simula **new activities**, none of which can be strictly classified as program or data. Its contribution to concise program specification arises because it allows some specific information to be bound, while leaving other information unbound. The "section" has been defined previously as a programming language construct [LR]. Several languages have a similar notation for implicit iterative control (FORTRAN IV, APL, PL/1), but do not permit the "section" in its full generality. Hence, we include it here as "innovative" to emphasize its importance as an idea which should be incorporated directly into existing programming languages.

4. Infinite sequence generation

The ability to define and operate on conceptually infinite sequences is the most obvious novelty in the language basis. Terminating a sequence external to its specification permits the effect. This idea is not new to data structures, where a pointer back to a previous element in a sequence may be interpreted by the program as a cycle. However, such a mechanism in programs is new†.

---

†Its utility relies significantly on the ability of a program to "represent" its result as distinct from "constructing" its result.

Programming experience with the basis has shown that the process of going from mathematical formulations of algorithms--in terms of polynomials, infinite series, etc.--to algorithms in the basis is aided significantly by the ability to represent unbounded sequences. From a more formal mathematical viewpoint, this is the first language in which it is possible to deal with the recursively-ennumerable sets directly.

5. Elimination of recursion

Any recursive language which permits the definition of functionals--functions with functions as arguments--has the facility for expressing what we have termed "recursionless constructs": operators which apply a function argument to other arguments recursively. For example, "maplist" in LISP applies a function to each of the elements of a list recursively.

The innovation in this work is (1) in identifying that such functions as "maplist" eliminate the need for explicit recursion in some cases, (2) in postulating that a "covering set" of such operators may exist which would ultimately permit the removal of all explicit recursion from languages, and (3) in providing examples of some rather powerful "recursionless constructs" which can be used in extant higher level languages. Although it may not be necessary to remove recursion from languages, it is important that we identify how recursion is and should be used, and then designate that activity with a language construct.

6. Correspondence between recursive data structure and control structure

Although the "recursionless constructs" we propose do not "cover" the common uses of recursion completely, we were able to show that recursive analogs to the "cosequential" operators can be defined and integrated into the language basis. The analogy is direct in the sense that "corecursive" operators were defined which emphasize the extent to which recursive data structures follow the recursive control structure of the functions which operate on them, or alternatively, the extent to which recursive functions follow the recursive data structures on which they operate. Although we were able to identify some quite powerful "corecursive operators", we are not convinced that they are fully adequate for the expression of desirable recursive effects. However, the indications are strong that the approach will be fruitful.

7. **Effects of subroutines in applicative languages**

The demonstration that coroutines can exist in a langauage without an assignment statement is innovative. That is, coroutines may be defined as an association of a "state" and a computation in that state, functionally dependent on other coroutines. The basis provides a method for expressing independent states and a method for relating them functionally. By allowing the effects of coroutines in an applicative language, we preserve the important equivalence property of applicative languages: identical expressions in the same static context have the same value.

None of the above ideas is extremely significant in isolation; the innovation of the basis derives principally from the ability of the above ideas in combination to reorient our approach to programming. In terms of traditional programming structures, the impact of the above ideas in combination is twofold:

1. By extending the representation traditionally used for control structures to data structures, we extend the **implicit** data structure representation. Although these data structures can be imposed explicitly by programs using data pointers or array subscripts, their implicit representation is significant to program conciseness.

2. Traditional data structures are accessed element-by-element by programs. Thus, the explicit dynamic relationships between a program and its data structures are very primitive. By emphasizing the relationship of sequences of accesses of data structures to the programs which perform the access, we have begun to structure the dynamic relationships between data and program. The operators accomplishing this dynamic structure are thus able to replace the traditional mechanisms for accomplishing these effects--namely, subroutines.

The primary languages which influenced the design of the basis were Bliss, APL, and LISP (in that order of importance). Their influence only becomes apparent after experience with programming in the basis. Relationships to other languages are similarly masked because of the reorientation of programming style the basis demands. The reorientation is not solely dependent on the absence of an assignment operator, but rather involves the necessity to recast formulations of programs to emphasize close correspondences between program and data structures. One quickly becomes cognizant of the extent to which the implementation of a sequence as a program or data depends on the context of the use of the sequence. Implicitly, one then recognizes the utility of specifying sequences independent of the context of their use. The basis forces one to restructure his approach to programming to emphasize the commonality of data structures and control structures and their relationships. This is the most important/innovative aspect of the basis, and it results from the combination of the ideas above.

Limitations

The limitations of the basis (as developed in this thesis) arise from two areas: aspects of programming languages orthogonal to nested-sequential representation, and reformulation issues in terms of applicative languages and nested sequences in general.

1. **The orthogonal elements of the basis**

The language basis was **a** priori constrained to describing interesting programming structures, through the use of nested-sequential structures only. We may characterize the remainder of the (semantic representation) space as the "unordered" or "parallelism" space, with orthogonal elements: sets, association mechanisms, parallel operations, type mechanisms, name spaces, etc. It is quite clear that the description of even nested-sequential structures is aided by elements from this space†. The basis has been pushed to the point where the interactions between parallelism and sequentiality should begin to be studied.

The basis is presently able to simulate parallel activity, but simulation of effects obtained easily in another representation indicates poor design when that representation is naturally implementable. For example, simulating **sub** is unrealistic, if a sequence can be recast as a parallel construct amenable to random access. An apparent alternative is to seek parallel implementation techniques for activities which are described as sequential. This is not a reasonable approach, for the choice of parallel vs. sequential implementation drastically affects the algorithm chosen for any particular task--parallel versions of efficient sequential algorithms are frequently less

- -

†The **case** gotoless construct is actually an element from this space, and is not present in the basis.

efficient (of time) than parallel versions of inefficient sequential algorithms.

To summarize, by leaving out considerations of parallel structure description, we have approached the extent to which sequential activity can be described solely in terms of itself. We do not suggest that parallel description is unimportant, even for sequential representations.

2. **Limitations of nested-sequential representation**

Naturally, forced sequential or recursive simulation of effects achieved best in a parallel representation--as through the use of sets or APL arrays--is not considered a limitation of this work. An adequate language basis must include the orthogonal elements mentioned above. Of more concern is the limitation of the pointerless representation for obtaining effects normally obtained using pointers.

We are faced with a problem in using the pointerless representation for data. Sometimes data must reflect a "real world" structure which may simply not be amenable to treatment as (potentially infinitely nested, cyclic) nested-sequences. Certain graphs cannot be adequately represented in this way, for example, and there are occasions when we do not have the freedom to impose the artificial gotoless representation. Although we have confidence in the "gotoless" constructs in control contexts, based on both formal and practical experience, we await future research along the lines developed above to establish a similar empirical base for the gotoless constructs applied to data structures.

Although many problems arising from the lack of an assignment statement are properly part of the parallelism domain (random access, for example), we cannot yet claim that all uses of assignment in traditional languages are preferably reformulated in the basis. The coroutine primitives may be used to study the extent to which we can define constructs which give the effects of assignment such as modification of data structures. We feel that more work in discovering such coroutineless constructs is required before the necessity for assignment can be characterized effectively.

Future Research

Work of this nature is successful solely to the extent that it is able to stimulate future research: we have not in any sense attacked a problem and solved it, but have rather presented a set of ideas and indicated how they are interrelated. The work is so open-ended that we hesitiate to eliminate any subfield of computer science as a candidate for its further development. However, there are three major areas which should pay attention to the ideas presented herein: language design, formal programming studies (program verification, structured programming, and formal semantics specification), and implementation studies (optimization and machine design).

The impact of this work on language design may take some time to emerge; the ideas in the basis are not easily factored from the basis in a manner directly applicable to improving existing languages. As we have mentioned, the "section" and the recursionless constructs may be useful in such a context, but it should be clear that existing languages must be significantly reformulated to incorporate most of the ideas in the basis.

The principal stimulation this work can provide to language design is to demonstrate that a fundamental reformulation of languages may be in order. Although we have spent significant effort demonstrating the evolution of the elements of the basis from conventional concepts, the impact of the basis is that it is fundamentally different from other languages. Continued research along the line of reasoning followed in the development of the basis is necessary: what other "coroutineless" constructs--both sequential and recursive--are desirable, what formulation of the parallelism space is appropriate, how do data structures, name spaces, type mechanisms, etc., impact the work? There are a large number of questions that only researchers with considerable programming experience can answer, dealing with the aptness of new constructs which should enter the basis. That is why the presentation has been so obviously informal and directed to the language design audience specifically.

This work may have considerable impact as a formal semantics specification language (after it is extended and formalized). Formal semantics should be specified as concisely as possible. They should also require as little "conceptual interpretation" as possible. The only distinction between the best programming language and the best formal semantics language should be that the semantics language is higher-level. It is considerably more difficult to specify how something should be built up than to demonstrate how it is a special case of something more general about which considerable knowledge has already been accumulated.

The impact of the basis on program verification and other formal approaches to programming should be considered. The techniques of Gerhart [GR] in verification studies of APL are probably more appropriate in this context than those of King [KI] and Hoare [HO]. In particular, one does not arrive at algorithms in the basis as easily by modifying variables in an invariant relation as he does deriving the algorithm directly from a mathematical model involving sequences. (See Appendix IV for an example of this phenomenon.) In fact, it is almost as difficult to understand the transformation of a traditional gotoless language algorithm into an algorithm in the basis, as it is to understand the analogous transformation from assembler language into a higher-level language.

Finally, implementation of the basis looks extremely interesting as a future research effort. The primary language from which the basis was derived was Bliss--it is very likely that compilation of programs in the basis is not too difficult. Naturally, by interpreting a representation as opposed to interpreting programs or data, we open the area of internal representation of programs and data--by the same token, we unify the approach. We strongly suggest the approach of Hansen [HA] to implementing the basis, optimizing only when necessary and only to the extent necessary. The ultimate goal of every language designer is to produce a machine for which the language is the machine language. Efforts in machine design such as the STAR VI [HT] are very promising as a technology for such an implementation.

Obviously, by defining an (unnamed) language basis, we are not interested in controlling the future research from the basis (although we would certainly be interested in hearing of any such efforts). We are particularly uninterested in defining a sequence of (upward compatible) languages from the basis, but encourage any reformulation appropriate to the research at hand. It is a rare opportunity for those interested in optimization efforts to be permitted to reorient a language to facilitate their effort--here is a basis for one.

Order of Magnitude Improvement

In Chapter 1 considerable attention was paid to finding an order of magnitude improvement in general purpose programming languages. Our only claim is that we feel a language derived from the basis may attain such a distinction. The lack of an association mechanism and other "parallelism space" desirables prevents a concrete demonstration of the claim. We can only summarize that the basis is presently significantly more concise than Algol for a larger class of problems than is APL, but it is not as concise as APL for the problems for which APL is particularly well-suited. This conciseness relies on the build-up of a considerable library of useful functions; however, we are far better able to rely on such a traditionally difficult entity because of our ability to represent infinite sequences and to deal with programs and data uniformly. Implementation does not appear to be a difficult task--for some programs in the language basis, efficiency can be commensurate with that of current languages.

In the last twelve years of language design research, the order-of-magnitude criterion has not been met for general purpose programming languages by pushing traditional language constructs. We feel it will only be met by making both the programmer and the implementation aware of higer-level relationships between program and data structures, and by emphasizing these relationships with language constructs facilitating their concise expression and efficient compilation. Expressing such relationships demands that we step outside traditional language structures. We believe that the basis represents a significant step in this direction.

APPENDIX I

INSTANTIATION, COMPOSITION AND EVALUATION NOTES

Instantiation rule: For b a binary operator and x an operand:

a. x b stands for the operator defined by:

```
none op y :: x b y;
```

b. b x stands for the operator defined by:

```
y op :: y b x;
```

c. x b y stands for the instantiation: x (b y).

Composition rule: If b, l and r are binary, left-unary, and right-unary operators, respectively:

a. b l stands for the binary operator defined by:

```
x op y :: x b (l y);
```

b. r b stands for the binary operator defined by:

```
x op y :: x r b y;
```

c. r_1 r_2 stands for the right-unary operator defined by:

x op :: x r_1 r_2;

d. l_1 l_2 stands for the left-unary operator defined by:

none op y :: l_1 (l_2 y).

The resulting operators are then subject to the composition rules. No other combination of operators is a composition (see next section).

Extensions to the composition rule:

The composition rules above explicitly disallow compositions of the forms: "b r", "l b", "r l", "l r" and "b b". In fact, the first two forms have reasonable interpretations--namely that the unary operator be applied after the operation and parameters be bound as though the unary operator were not present. That is, the above composition rules could be extended to permit:

e. b r stands for the binary operator defined by:

```
x op y :: x b y r;
```

f. l b stands for the binary operator defined by:

```
x op y :: l (x b y).
```

The forms "r l", "l r" and "b b" could be used to allow parameters to enter the expression from either side, but is rejected as nonintuitive and presently considered in error.

Impact of evaluation function on composition rules

Reference is made in Chapter II to the ambiguity of permitting:

```
+* == <+; +; ...>
```

but not allowing:

```
+ (mul 3).
```

Indeed, the composition rule (e.) above would permit this latter operation, but the result would be inconsistent with that of "+*". This points up the fact that an evaluation function will determine when the composition rules are to be invoked and when an operator can be applied to another operator directly.

Quote rules and the ability to define "functionals" (functions which permit functions as arguments) are normally used to resolve such an ambiguity. In the text, the "patterned expansion" of the functions has been used to convey the intended choice. Quote rules always cause problems when several levels of quoting occur. We see no solution to this problem, but do make the following "notes":

1. Permitting the ability to quote an argument at the operator definiton site is desirable. For example, assume "'" preceeding an argument name in an operator definition (to the left of the "::") indicates that when the operator is called, the corresponding actual parameter may be an unevaluated function. Assume also that "'" in an expression inhibits evaluation of a function, and indicates that the argument is to be considered "data"--i.e. directs the evaluation function to apply the function instead of compose the two. Then the distributed usage of the function throughout the program does not require that the argument-function be quoted in each instance. I.e.,

   ```
   'a f b :: b . (a*)
   ```

   does not require

   ```
   (...'+ f b1 ...  '- f b2 ...  'mul f b3 ...  etc.)
   ```

   but rather permits the same effect using

   ```
   (...  + f b1 ...  - f b2 ...  mul f b3 ...  etc.).
   ```

   The designers of LISP recognized this when defining "setq" for example, but did not permit the user to define such functions.

2. The ability of an operator to quote its argument may be inferred from its usage in its defining expression (by an interpreter or compiler). In the example above, given that "*" can quote its argument, it is redundant to specify the fact explicitly using the "'".

3. The "*" is already a form of quote operator (see Chapter III: A **recursive** "*".)

APPENDIX II

NOTES ON THE MULTIVALUED-LOGIC OPERATORS

In Chapter II it was indicated that the operators **then, else** and **excludes** have somewhat anomalous properties. These stem from the interpretation of any non-nil value as "true" and nil as "false". In particular, if D is the (a) domain of the logic operators, and O the set of such operators (called connectives), then for no o in O is there an element t in D - {nil} such that: nil o nil = t.

We first consider such a logic for the domain {0,1}, where "0" is an abbreviation for nil. The set of connectives for this domain is called O'; they are ennumerated in Table AII.1 below.

o' in O'†		**null**	**and**	**x-y**	**lid**	**y-x**	**rid**	**xor**	**or**
x	y								
0	0	----	----	----	---- 0 ----	----	----	----	----
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

TABLE AII.1: Restricted Boolean Connectives, O'.

Notice the absence of the "exotic" connectives such as **nand** and **nor**.

We can now define the logic system of the basis as the set of binary mappings o in O from D x D into D subject to the following constraints:

1. For all o in O, nil o nil = nil;

2. For all x, y in D, x o y is in {nil, x, y};

- -

†lid is the binary left identity function, rid is the binary right identity function, and the operator "-" is actually "monus".

3. For each o in O, there must be an o' in O' such that the mapping b defined by

   b(x)=1 if x is in D - {nil}
   =0 if x = nil,

   is a homomorphism. That is:

   b(x o y) = b(x) o' b(y).

These latter two constraints remove domain dependencies from the connectives. In particular, (2) eliminates a connective which maps (x,nil) onto y, and (3) eliminates connectives which map (x,nil) onto x but (y,nil) onto nil (for x,y in D-{nil}). The table below represents all such connectives† (with "0" substituted for nil):

x	y	Connectives											
0	y	0	0	0	0	0	0	y	y	y	y	y	y
x	0	0	0	0	x	x	x	0	0	0	x	x	x
x	y	0	x	y	0	x	y	0	x	y	0	x	y
Ref. #		1	2	3	4	5	6	7	8	9	10	11	12
Partitions		---	--------		---	--------		---	--------		---	--------	
names††		null	and		x-y	lid		y-x	rid		xor	or	
Converses		1	3	2	7	9	8	4	6	5	10	12	11

TABLE AII.2: "Language Basis" connectives, O.

- -

†To be precise we would have to define a homomorphism from an arbitrary D onto {x, y, nil}, etc.

††The homomorphism b induces a partition on O. In particular, o and p in O are in the same partition iff for all x, y in D, b(x o y) =b(x p y). The corresponding element of O' is given here as a name for the partition.

Completeness Properties

This logic system has some rather interesting "completeness properties". A set of connectives, S, will be called complete iff each connective o in O can be written as a well formed expression in terms of x, y and the elements of S.

The properties of interest here are:

1. {**excludes, else**} is a complete set of connectives;
2. There is no complete unitary set of connectives;
3. {**excludes, then**} is not a complete set of connectives.

In Chapter II the correspondences between **excludes** and **not**, **then** and **and**, and **else** and **or** were made. Thus, the properties (2) and (3) may seem somewhat startling, in terms of regular Boolean logic. The proofs of these properties are sketched below.

Proof Sketches

1. {**excludes, else**} is a complete set of connectives.

Pf. First notice that we need not concern ourselves with converses (simultaneously substitute x for y and y for x in an expression to obtain the converse of the connective which the expression defines). Also note that the left and right identities, lid and rid, respectively, can be obtained directly (e.g. for "exp **lid** y" substitute "exp"). Hence, we need only construct the connectives with reference numbers (in Table AII.2) of 1, 3, 6 and 10:

x (1) y = x **excludes** x;
x (3) y = (x **excludes** y) **excludes** y;
x (6) y = ((x **excludes** y) **excludes** y) **else** x;
x (10) y = (y **excludes** x) **else** (x **excludes** y).

QED

2. There is no complete unitary set of connectives.

Pf. Assume there is, and assume the set is {c}. Then the corresponding Boolean connective c' must be such that {c'} is a complete set for O' (c' is the label of the partition of which c is a member in Table AII.2). In fact, there is no complete unitary subset of O'!

None of the following sets is complete: { - , **null**, **lid**, **rid**, **and**}, {**xor**, **null**, **lid**, **rid**}, {**or**}. Hence, no proper subset of these sets is complete. Since each of the connectives appears in at least one of these sets,no unitary set is complete. (The sets arise from generating all expressions involving "-", **xor**, and **or**, respectively.) Hence, there is no complete unitary subset of O.

QED

3. The set {**then**, **excludes**} is not complete.

Pf. The proof of property 1 expresses then in terms of **excludes** (alone). This set is complete, therefore, iff {**excludes**} is complete, which it is not, by property 2.

QED

APPENDIX III

USEFUL FUNCTIONS DEFINED IN THE TEXT

Note: many of these functions can be simplified to be defined in terms of other useful, more primitive functions. We do not propose this set as a well-integrated set of functions, but include the list merely for reference from within the text. If an operator is redefined, each version appears here, in the order of redefinition in the text. Most operators which are used only in a very local context in the text are not redefined here.

Function	Page Defined
and :: **then**	35
x alternate y :: <x; y> **gen** *	36
M Bmask :: M . (controls (1*) *)	54
s conc p :: <s **gen**; p **gen**>	34
s controls q :: s . (rid *) . q	39
s controls q :: s length lt (q length) **excludes** (s controls q)	44
s controls q :: s controls (q lengthge (s length))	44
s controls q :: s . (rid *) . (q conc (nil *))	45
M column i :: M . (sub i *)	51
s eo :: s. (<**emitn** odd;**emitn** even> **gen** *) : [even; odd]	97
x exch :: <x **emit**; **collect**> **val**	103
M fromnil :: M . (. (**else** NIL *) *)	53
M fromNIL :: M . (. (**ne** NIL *) *)	54
n factorial :: 1/(mul * . (n pos)) **val**	59
n head s :: n **gt** 0 **then** (n pos controls s) **else** <>	42
x id :: x	27
u ip v :: u rowmul v sigma	51
s interleave p :: [s::s; p::p] : <**collect** s; **collect** p> **gen** *	99

s length :: s controls P val else 0	44
s lengthge i :: i pos controls s length = 1 then s	44
i max j :: i ge j else j	48
M MM N :: M . (rM N *)	51
s mask :: s . (thenf emit *)	95
none not x :: x excludes true	35
s nonempty :: s . (exs *)	40
v notempties :: v . (txs *) val then v	52
x null :: <> gen	63
or :: else	35
k odd :: k mod 2 = 1	57
P :: 0/(+1*)	41
n pos :: P while (le n)	42
s prefix y :: <y> conc s	57
s prefix y :: <y; s gen>	107
x rid y :: y	27
s rplus p :: s . (+ *) . p	39
u rowmul v :: u . (mul *) . v	51
r rM M :: r ip * . (M transpose)	51
s reverse :: <>/ (prefix * . s) val	57
fs reduce s :: fs functionzero / (fs . s)	77
fs reduce s :: / (fs . s)	78
fs reduce s :: fs 1st / (fs tail . s)	78
s sub i :: i head s val	43
u sigma :: 0 / (+* . u) val	51
M transpose :: M column * . P	51
M transpose :: M column *. P while notempties	52
M transpose :: M fromnil transpose fromNIL	54
M transpose :: M Bmask transpose controls (M column *. P)	55
list tail :: list . <null; id * gen>	63
x thenf f :: x then (x f)	95
x whenf f :: x f then x	(Appendix V)

s while f :: s . (f exs *)	42
list 1st :: list . <id> val	63
list 2nd :: list . <null; id> val	(Appendix V)
list 3rd :: list . <null; null; id> val	(Appendix V)

APPENDIX IV

COMPILING EXPRESSIONS IN THE INITIAL BASIS

To indicate that we are indeed very serious about the basis as a **realistic** approach to programming, the following section indicates how compilation is possible for some expressions in the basis. A fairly complex example is worked out in considerable detail, producing a very efficient program (which could seem unlikely to one who is seeing the basis for the first time).

To illustrate, Knuth's "Algorithm A" for computing x raised to the power n (originally Legendre's algorithm) [KN, pp. 399-400] is compiled. A few words about the algorithm and formulation in the basis are in order before the compilation process is indicated. The algorithm essentially arises from the equivalence: (using "↑" to indicate exponentiation here)

$$\begin{aligned} x \uparrow n &= x \uparrow (d_1 + 2\ d_2 + 4\ d_3 + \ldots\ + (2\uparrow(i-1))\ d_i) \\ &= (x \uparrow d_1)\ ((x \uparrow 2) \uparrow d_2) \ldots\ ((x \uparrow (2 \uparrow (i-1))) \uparrow d_i) \end{aligned}$$

where the "d_i" are the coefficients in the binary expansion of "n".

When d_i=0, the term in the product above is "1". That is,

$$(x \uparrow (2 \uparrow k)) \uparrow 0 = 1$$

and

$$(x \uparrow (2 \uparrow k)) \uparrow 1 = x \uparrow (2 \uparrow k).$$

Thus, the algorithm simply involves computing factors involving successive squares of "x".

To represent the algorithm in the basis, we first note that the squared powers of "x" are:

```
x squared :: x mul x;
x power2  :: <x; x/ (squared *) gen>                          [1]
```

The binary coefficients in right-to-left order can be obtained:

```
    n bincoef :: <n; n/(div 2 *) gen> while (ne 0) . (mod 2   *)           [2]
```

The factors to be multiplied can be written:

```
    x factors n :: n bincoef . (=0 then 1 else *) . (x power2)              [3]
```

Thus, the algorithm for "x↑n" can be written:

```
    x tothe n :: 1/(mul * . (x factors n)) val                              [4]
```

In compiling programs in the intial basis we deal with three separate program representations; the source, an intermediate representation which we call "generator expressions" and the target language, here a dialect of Bliss [WU,1972].

Generator expressions will be objects from which we can "collect" a value. The objects will be triples:

G = [[L; D; E]].

(The double brackets are used to avoid confusion with the coroutine set notation of Chapter IV.) A generator is such that a program counter or sequence position counter can be associated with it. The portions of the generator are:

L(G) = label set;
D(G) = declaration/ initialization set;
E(G) = generating expression.

The label set arises from escapes in sequences expressions. In translating from the basis to generator notation, several levels of sequencing operations will be merged--i.e., the escaped sequence would become ambiguous unless we tied it to a unique label. The declaration sets arise from "accumulate" operations where a temporary variable must be declared to accumulate the result. The expressions, E, will be quite similar to expressions in the basis defined solely in terms of the primitive functions, except they will involve the declared variables of the "declaration/initialization set", and assignment, "←".

We will be concerned with when we can translate a generator expression into either a subroutine with **own** variables--i.e. a coroutine--and when we are able to convert the expression to a closed function with local variables. Below we introduce rules used to transform sequence expressions from the initial basis into the intermediate generator notation from which either subroutines or functions can be compiled.

T1: e * ==> [[{L};; **cycle** e']]

where L is a unique label, and e' has all **exs** expressions redefined as **exsn** L. L need only be defined when such an exit exists in e.

T2: [[L; D; **cycle** E]] . [[L'; D'; **cycle** E']] ==>
[[L **union** L'; D **union** D'; **cycle** (E (E'))]]

T3: x / [[L; D; **cycle** E]] ==>
[[L; D **union** {**declare** v = x}; **cycle** (v←v(E))]]

where v is a unique name, not in the program.

T4: ([[L; D; E]]) ==> [[L; D; (E)]]

The transformations are to be applied "inside out" and "left to right" to expressions in the initial basis. They transform primitive functions (arithmetic, relational, conditional) and expressions composed only of primitive functions intact.

We now consider the compilation of the "x↑n" algorithm above. However, to circumvent issues involving the **gen** operator, we redefine "power2" and "bincoef" in the following somewhat artificial way†:

```
x power2 :: x sqrt / (squared *)                    [5]
x bincoef :: 2 mul n / (div 2 *)                    [6]
             while (ne 0) . (mod 2 *)
```

- -

†Note, we can replace <v; v/(f*) **gen**> by "v (f inverse)/ (f*)" when "f" has a unique inverse. In general, we can replace it by: "<1;v>/(2nd/<id;f>*).(1st*)". This latter expression corresponds to the normal nasty situation where a side-effect must occur, but the previous value of the changed variable is desired after the assignment.

We begin by applying the transformations above to the subfunctions of [4].

```
x power2 :: x sqrt/(squared*)

   T1 ==> x sqrt/[[;;cycle squared]]
   T3 ==> [[; {declare Z=x sqrt}; cycle (Z←Z(squared))]]                   [7]


n bincoef :: 2 mul n/(div 2*).((ne 0) exs*).(mod 2*)

   T1 ==> 2 mul n/[[;;cycle div 2]].(ne 0 exs*).(mod 2*)
   T3 ==> [[; {declare N=2 mul n}; cycle (N←N(div 2))]].
          (ne 0 exs *).(mod 2 *)
   T1 ==> [[; {declare N=2 mul n}; cycle (N←N(div 2))]].
          [[{DONE};; cycle (ne 0 exsn DONE)]] .(mod 2 *)
   T1 ==> [[; {declare N=2 mul n}; cycle (N←N(div 2))]].
          [[{DONE};; cycle (ne 0 exsn DONE)]] .
          [[;; cycle (mod 2 *)]]
   T2 ==> [[{DONE}; {declare N=2 mul n};
           cycle ((N←N(div 2)) (ne 0 exsn DONE))]].
          [[;; cycle (mod 2 *)]]
   T2 ==> [[{DONE}; {declare N=2 mul n};
            cycle ( ((N←N(div 2)) (ne 0 exsn DONE)) (mod 2) )]]          [8]
```

Note also that we can transform "(=0 then 1 else *)" by "T1" to:

```
   [[;;cycle (=0 then 1 else)]]                                           [9]
```

We can then do "factors" using [7], [8], and [9]:

```
x factors n :: [8] . [9] . [7]

   T2 ==> [[{DONE}; {declare N=2 mul n};
            cycle ( ( ((N←N(div 2)) (ne 0 exsn DONE)) (mod 2))
                    (=0 then 1 else) )]] .
         [[; {declare Z=x sqrt}; cycle (Z←Z(squared))]]

   T2 ==> [[ {done};
            {declare n=2 mul n; declare Z=x sqrt};
            cycle (((((N←N(div 2))(ne 0 exsn DONE))(mod 2))
                (=0 then 1 else))
                (Z←Z(squared)) )]                                      [10]
```

Now, using [10] in [4] we can transform the exponentiation function:

```
    x tothe n :: 1/(mul * .  [10]) val

      T1 ==> 1/([[;; cycle mul]] .  [10]) val
      T4,T2 ==> 1/ [[ {DONE};
            {declare n=2 mul n; declare Z=x sqrt};
            cycle (mul (((((N←N(div 2))(ne 0 exsn DONE))(mod 2))
                 (=0 then 1 else))
                 (Z←Z(squared)) )]] val
      T3 ==> [[ {DONE};
            {declare n=2 mul n; declare Z=x sqrt; declare Y=1};
            cycle (Y←Y( (mul (((((N←N(div 2))(ne 0 exsn DONE))
                 (mod 2))(=0 then 1 else))
                 (Z←Z(squared)) ) )]] val                             [11]
```

Using the instantiation rules of Appendix I, we can remove parentheses to obtain:

```
    x tothe n :: [[ {DONE};
            {declare N=2 mul n; declare Z=x sqrt; declare Y=1};
            cycle Y←Y mul
                    (((N←N div 2) ne 0 exsn DONE mod 2)=0
                       then 1
                       else (Z←(Z squared))) ]] val                   [12]
```

The expression inside the cycle should look at least reasonably close to a "real" program for "x tothe n". (The names of the variables are consistent with those used in Knuth's version of the algorithm.) Notice that the computation sequence (cycled expression) could be "pulsed" if there were occasion to do so, by producing a program from the generator expression translating declare into own in Bliss or Algol. However, the val operation indicates that the variables of the generator are temporary in nature and hence, that the declares can be local declarations in Bliss. We can convert the above program into a Bliss program almost trivially:

```
routine tothe(x,n) =
  begin
    local N=2 * .n, Z=sqrt(.x), Y=1;
    label DONE;
    DONE : while true do
        Y←.y * if
                (if (N←.N/2) eq 0
                    then leave DONE
                    else .N mod 2 eq 0)
            then 1
            else (Z←.Z * .Z);
    .Y
  end;
```

[Note: in Bliss, "." takes the contents from a machine address, which declarations associate with declared variables. The **leave** expression escapes from the expression labelled by its argument. The value of a block is the last expression in the block--".Y" in this case.]

Although this expression is not optimal, normal optimization will transform the multiplication so it only occurs on odd values of "N". Bliss will even do a right shift for the divide and a mask operation for the test.

The expressions for the initial values of X and Z are discomforting. We claim (without proof) that they could be hanndled in a better way by using the somewhat obscure formulation in the footnote above, or, in fact, by the proper considerations of **gen**.

We will not present any more compilation issues here: the above discussion is intended to indicate that we do not feel that the basis is even as unrealistic as LISP in terms of compiling efficient programs. Some efficient programs can be compiled with an almost trivial amount of optimization effort.

Our considerations are far from complete; the mechanism above may have to be modified drastically to accomodate the other operations in the basis. In addition, we have ignored issues of parameter substitution mechanisms (used implicitly in the transformed expression above), data structure creation, nested loops, etc. Such considerations should await a formalization of the basis: both formalization and compilation constitute significant research efforts in themselves.

However, one point is extremely important: the compilation considerations above are possible only because "." and "/" have been identified as primitive coroutineless constructs. Had they been extended from "collect" and "emit" as is suggested in Chapter IV, optimizers might have missed the transformations above and not compiled as efficient programs. The analogy is direct between "coroutineless" and "gotoless" programs: each presents a set of constructs whose interrelationships can be considered by implementers to produce well optimized programs. If the constructs are not present, the optimizer is unable to confine his attention to the most frequent functional usage of the goto or coroutine call. He will probably not be able to focus on the specific cases above because of the interference of the uses of the primitive constructs. That is, he must recognize the use of "." and "/" by "pattern match", insure other coroutine calls do not interfere, and then apply the transformations.

APPENDIX V

COMPUTATIONAL COMPLETENESS OF THE INITIAL BASIS

Computability

In order to prove that the initial basis (described in Chapter II) is computationally complete, we show that an arbitrary Turing Machine can be implemented using the basis. Thus, in particular, a universal Turing Machine can be simulated, and the partial recursive functions are computable in the basis. The terminology follows Hopcroft and Ullman [HU].

A Turing Machine is a finite state device with a semi-infinite tape on which symbols from an alphabet, GAMMA, can be written and from which they can be read. The set of states will be called, K. A Turing Machine instruction, called a "move", determines the next configuration of the machine by specifying:

1. The next state;
2. The symbol to be written on the current position of the tape (under the read/write head);
3. The direction the tape must be moved--left, L, or right, R.

A move depends on the current symbol under the tape head and the current state. A program (set of moves) must be specified by a function,

delta: K x GAMMA --> K x (GAMMA-{B}) x {L, R}

where B (blank) is the symbol in any tape position not yet scanned (read) by the machine. A computation proceeds one move at a time, until a state in the final state set, F (a subset of K), is reached. The non-blank portion of the tape is the result of the computation.

Initially, a machine is started in state q_0, with a sequence of symbols A_1, A_2, ..., A_n on the tape. The head is positioned at the leftmost symbol (A_1) and the remainder of the tape is blank (all Bs).

To implement a Turing Machine in the basis, the function delta, the initial state q_0, the final state set F and the argument sequence A must be provided. (These latter two sets are specified as sequences in the basis with the same names--i.e. F and A.) We define an intantaneous description of the computation (similar to Hopcroft and Ullman's "TM configuration") as the following sequence:

```
<current state--"q";
  <Tape to the left of the head, reversed--"LH";
   Symbol under the head--"h";
   Tape to the right of the head--"RH">>
```

Thus, if the machine is in state "b", the tape has the symbols "xyzpqBB...", and the head is positioned on "z", the instantaneous description is:

```
<b; <<y; x>; z; <p; q; B; B; ...>>>
```

The various fields of an instantaneous description, ID, may be accessed by the functions defined below:

```
q :: 1st; h :: 2nd 1st; LH :: 2nd 2nd; RH :: 2nd 3rd.
```

The function delta's result is formatted†,††:

```
q delta g ==
  <next state--"q'"; written symbol--"g'"; head direction--"LorR">
```

where q is the current state and g is the symbol under the head. Accessors for values of this function are defined:

```
q' :: 1st; g' :: 2nd; LorR :: 3rd.
```

- -

†Technically we must show the basis is able to express arbitrary "delta" functions. By naming the states with positive integers, and a GAMMA of the decimal digits (**union** {B}), an array of triples in the above format may be simply selected to produce the result of delta.

††We prove that the initial basis **including an operator definition facility** is complete. This differs from lambda-expressions [CH], for example, where a universal function may be expressed as a closed expression in the system.

The implementation essentially relies on a two stack machine simulation of the TM computation. A stack is stored as a sequence, the first element of which is its "top". The relevant operations on stacks are:

```
stack push x :: <x> conc stack;
stack pop :: stack tail;
stack top :: stack 1st.
```

A "move" function which transforms one instantaneous description to the next may then be written:

```
ID move :: ID q delta (ID h) MOVE ID
```

where MOVE is defined:

```
D MOVE ID ::

   <D q';
    ID * .  (D LorR = "L"
        then <LH pop; LH top; RH push (D g')>
        else <LH push (D g'); RH top; RH pop>)>
```

The computation sequence may then be described as:

```
COMP :: <q0; <<>; <A 1st else B>; A tail conc (B *)>> / (move *)
```

However, the above computation does not terminate. To obtain the finite computation sequence (when there is one), the following auxilliary functions are useful:

```
x isnotin s :: s .  (= x txs *) val excludes x;
x whenf f :: x f then x.
```

The terminating computation sequence is then simply:

```
COMP' :: COMP while (whenf (q isnotin F))
```

To obtain the value of the computation we simply decode the last ID in COMP':

```
ID decode :: ID LH reverse conc <ID g> conc (ID RH) while (ne B);
TMCOMP :: COMP' val decode.
                  Q.E.D.
```

Diagonalization

Perhaps a more interesting effect in the basis is the ability to deal with recursively ennumerable sets directly. To demonstrate, we express a function whose value is the dovetailed computation of all TM computations on a blank tape.

We postulate a generator for "deltas" which generates all 2-dimensional arrays of triples conforming to the rule that the computation begin in state "1" (q0 = 1) and terminate in state "2" (F = {2}), if it terminates. (Obviously, there is no loss of generality here.) Call the generator for the deltas, DELTA--assume each element of DELTA is a two-dimensional array of triples such that the first subscript (the rows) correspond to states, and the second subscript corresponds to tape symbols from a fixed alphabet, the decimal digits. We interpret "0" as "B" and disallow it from being written.

Then for DEL an element of DELTA, we define:

```
args del DEL :: DEL sub (args 1st) sub (args 2nd)
```

(where args is a two element sequence <current state, current symbol>). The functional (del DEL) then represents a valid "delta" with its arguments encoded. That is, we can redefine "move" as:

```
ID move DEL :: <ID q; ID h> del DEL MOVE ID
```

In particular, we can now define the (blank tape) computation sequence of a Turing Machine in DELTA as:

```
BTC DEL :: <1; <<>; 0; 0*>>/ (move DEL*)
        while (whenf (q ne 2))
```

where MOVE is as above.

All blank tape computations can then be described as:

```
ALLBTC :: BTC * .  DELTA
```

Obviously, we must be rather careful how we access this monster. We cannot ask for the value of the first computation and hope to do anything with the second. Frequently, such sequences are considered, however, and "dovetailing" is used to describe a desired effect. If we arrange the computations in the following way, it will be clear how such a process works:

```
<<C11; C12; C13; C14; ...                  >;
 <nil; C21; C22; C23; ...                   >;
 <nil; nil; C31; C32; ...               >;
              ...
              ...
              ...                                  >
```

The empty elements have been introduced as "place holders" for the dovetailing process. Dovetailing involves taking the columns of the doubly infinite array described above (ALLBTC), until an empty element is encountered in the column. Equivalently, we can take one element from column 1, 2 elements from column 2, 3 from 3, etc. Thus, we can define a dovetail function for any two dimensional infinite array as:

```
none convert A :: <0; P gen> .  (head (nil *) *) .  (conc *) .  A

A DOVETAIL :: P .  (head *) .  (convert A)
```

(remember transpose works for such arrays; must check that "0 head s == <>")

The dovetailed blank tape computations are then:

```
ALLBTC DOVETAIL
```

Noticing that the dovetailed array's rows increase in length for each successive row, and that nil will be the value after a computation halts, we can ennumerate the index in DELTA of the machines that halt (redundantly, here) by "HALTING" below:

```
row halt :: row .  (excludes *) .  P mask
HALTING :: ALBTC DOVETAIL .  (halt gen *)
```

APPENDIX VI

COROUTINE CONTROL EXAMPLE

Below, the evaluation sequence for the coroutine expression [IV, 12] is presented. The symbol "%" represents the current sequencer position in each of the coroutines. The "#" indicates the current "program counter", the point at which control actually resides (see [IV, 11] ff.). Assume control is initiated at "b", as would occur if that sequence were explicitly selected.

```
<collectn a emitn b; collectn b emitn a> * :
[a :: <3; <1 emit; collect> val; 4; <2 emit; collect> val>;
 b :: <# <100;101> sub collect emit; <200;201> sub collect emit>]

<collectn a emitn b; collectn b emitn a> * :
[a :: <3; <1 emit; collect> val; 4; <2 emit; collect> val>;
 b :: <<100;101> sub # collect emit; <200;201> sub collect emit>]

< # collectn a emitn b; collectn b emitn a> * :
[a :: <3; <1 emit; collect> val; 4; <2 emit; collect> val>;
 b :: < <100;101> sub % collect emit; <200;201> sub collect emit>]

<collectn % a emitn b; collectn b emitn a> * :
[a :: <# 3; <1 emit; collect> val; 4; <2 emit; collect> val>;
 b :: < <100;101> sub % collect emit; <200;201> sub collect emit>]

<collectn % a emitn b; collectn b emitn a> * :
[a :: <3; <1 # emit; collect> val; 4; <2 emit; collect> val>;
 b :: < <100;101> sub % collect emit; <200;201> sub collect emit>]

<collectn a # (= 1) emitn b; collectn b emitn a> * :
[a :: <3; <1 %; collect> val; 4; <2 emit; collect> val>;
 b :: < <100;101> sub % collect emit; <200;201> sub collect emit>]

<collectn a emitn # b; collectn b emitn a> * :
[a :: <3; <1 % emit; collect> val; 4; <2 emit; collect> val>;
 b :: < <100;101> sub % collect emit; <200;201> sub collect emit>]

<collectn a emitn % b; collectn b emitn a> * :
[a :: <3; <1 % emit; collect> val; 4; <2 emit; collect> val>;
```

```
  b :: < <100;101> sub collect # (= 1) emit; <200;201> sub collect emit>]

 <collectn a emitn % b; collectn b emitn a> * :
 [a :: <3; <1 % emit; collect> val; 4; <2 emit; collect> val>;
  b :: <100 %; <200;201> sub collect emit>]

 <collectn a emitn b; collectn b # (= 100) emitn a> * :
 [a :: <3; <1 % emit; collect> val; 4; <2 emit; collect> val>;
  b :: <100 %; <200;201> sub collect emit>]

 <collectn a emitn b; collectn b emitn # (= 100) a> * :
 [a :: <3; <1 % emit; collect> val; 4; <2 emit; collect> val>;
  b :: <100 %; <200;201> sub collect emit>]

 <collectn a emitn b; collectn b emitn % (= 100) a> * :
 [a :: <3; <1 emit; collect # (= 100)> val; 4; <2 emit; collect> val>;
  b :: <100 %; <200;201> sub collect emit>]

 <collectn a emitn b; collectn b emitn % (= 100) a> * :
 [a :: <3; 100; 4; <2 # emit; collect> val>;
  b :: <100 %; <200;201> sub collect emit>]

 <collectn # (= 2) a emitn b; collectn b emitn a> * :
 [a :: <3; 100; 4; <2 % ; collect> val>;
  b :: <100 %; <200;201> sub collect emit>]

 <collectn a emitn # (= 2) b; collectn b emitn a> * :
 [a :: <3; 100; 4; <2 % ; collect> val>;
  b :: <100 %; <200;201> sub collect emit>]

 <collectn  a  emitn  %  (=  2)  b;  collectn  b  emitn  a>  *  :          [1]
 [a :: <3; 100; 4; <2 % ; collect> val>;
  b :: <100 #; <200;201> sub collect emit>]

 <collectn a emitn % (= 2) b; collectn b emitn a> * :
 [a :: <3; 100; 4; <2 % ; collect> val>;
  b :: <100; <200;201> sub collect # (= 2) emit>]

 <collectn a emitn % (= 2) b; collectn b emitn a> * :
 [a :: <3; 100; 4; <2 % ; collect> val>;
  b :: <100;201 emit #>]
```

```
<collectn a emitn b; collectn # (=201) b emitn a> * :
[a :: <3; 100; 4; <2 % ; collect> val>;
 b :: <100;201 % >]

<collectn a emitn b; collectn b emitn # (= 201) a> * :
[a :: <3; 100; 4; <2 % ; collect> val>;
 b :: <100;201 % >]

<collectn a emitn b; collectn b emitn % (= 201) a> * :
[a :: <3; 100; 4; <2 # ; collect> val>;
 b :: <100;201 % >]

<collectn a emitn b; collectn b emitn % (= 201) a> * :
[a :: <3; 100; 4; <2; collect # (= 201)> val>;
 b :: <100;201 % >]

<collectn a emitn b; collectn b emitn % (= 201) a> * :
[a :: <3; 100; 4; 201 # >;
 b :: <100;201 % >]
```

At this point termination of "a" must cause control to resume in "b" and then pass the last element ("201") to the caller of "b". Note, the caller of "b" would have received the first element of "b" at the point marked [1], after the return from the **emit**. This is the point ([1]) when the value of emit is defined.

APPENDIX VII

RECURSION ORTHOGONAL TO SEQUENTIALITY

The functions below implement the "bottom up" function:

```
t bu f :: t atom? else (t .  ('bu f *) f)
```

Preliminary functions:

```
   a :: 1st
   v :: 2nd
   top :: 1st
   stack push x :: <x> conc stack
   stack pop :: stack tail
```

Top level function:

```
t bu f :: <<t>; <<>;<>>> / (g *) val v top 1st
```

where

```
s g :: s a notempty exs
        top empty then
           <s a pop; s v pop pop push
               (s v pop top conc <s v top f>)>
        else
        s a top 1st atom? then
           <s a pop push (a top tail);
            s v pop push (s v top conc <s a top 1st>)>
        else
           <s a pop push (a top tail) push (a top 1st);
            s v push <>>
```

APPENDIX VIII

IMPLEMENTATION OF GEN IN THE INITIAL BASIS

For a potentially unbounded sequence, s, and an index, i, the following functions define the sequence s with it ith element "gened"--e.g.

```
<1; <3;6>; 7> genf 2 ≡ <1; 3; 6; 7>
```

To obtain the actual function, both s and i must be encoded in a second sequence argument to f. They are passed as globals here for "clarity".

```
s genf i :: <nil; 1; 0> / (f *) .  (1st *)
```

where

```
q f :: q 2nd = i then (s sub i lengthge (q 3rd + 1))
   then <s sub i sub (q 3rd + 1); q 2nd; q 3rd + 1>
   else
      (s lengthge (q 2nd) exs
      then <s sub (q 2nd); q 2nd + 1; 0>)
```

(The first element of q is the element of the result sequence. The second element is an index for s. The third is an index for s_i when the second element is equal to i.)

Note, the **sub** function is the first function defined in Chapter 2:

```
s sub i :: i pos .  (rid*) .  s val
```

None of pos, rid and lengthge is defined using **gen**.

BIBLIOGRAPHY

[AB] Abrams, P. S. **An APL Machine**, Ph.D. Thesis, Stanford Univ., 1970.

[AJS] Adams, J. M., Johnston, J. B., Stark, R. H. (ed.) **SIGPLAN/SIGACT Proc. of an ACM Conference on Proving Assertions about Programs** (Jan. 1972).

[BA] Backus, J. W., et al. The FORTRAN Automatic Coding System. **Proc. of the Western Joint Computer Conference**, vol. 11, pp. 188-198, 1957.

[BA] Backus, J. **Reduction Languages and Variable-free Programming.** IBM Research Report RJ1010, 1972.

[BA] Backus, J. **Programming Language Semantics and Closed Applicative Languages.** IBM Research Report RJ1245, 1973.

[BH] Brinch Hansen, P. Structured multiprogramming. **Comm. ACM** 15,7 (July 1972), pp. 574-578.

[BN] Barnes, G. H., et al. The ILLIAC IV computer. **IEEE Trans.** C-17, 8 (Aug. 1968), pp. 746-757.

[BR] Buxton, J. N., and Randell, B. Portability and adaptibility (discussion). **Software Engineering Techniques** (April 1970), pp. 28-41.

[CH] Church, A. The calculi of lambda-conversion. **Annals of Mathematical Studies**, No. 6 (Princeton Univ. Press, Princeton, 1941).

[CG] Constable, R. L. and D. Gries. On classes of program schemata. **SIAM J. Computation** 1, 1 (1972).

[CF] Curry, H. B. and Feys, R. **Combinatory Logic** Vol 1 (North-Holland, Amsterdam, 1968).

[DMN] Dahl, O. J., Myhrhang, B., and Nygaard, K. SIMULA 67 common base language. Publ. # S-2, Norwegian Computing Center, Oslo (May, 1968).

[DN] Dahl, O. J., and Nygaard, K. SIMULA--an ALGOL-based simulation language. **Comm. ACM** 9,9 (Sep. 1966), pp. 671-678.

[DI] Dijkstra, E. W. Goto statement considered harmful. Letter to the editor. **Comm. ACM** 11, 3 (Mar. 1968).

[DI] Dijkstra, E. W. The structure of the "THE"--multiprogramming system. **Comm. ACM** 11, 5 (May 1968), pp. 341-346.

[DI] Dijkstra, E. W. Structured programming. **Software Engineering** (Oct., 1969, Rome).

[FGP] Farber, D. J., Griswold, R. E., and Polonsky, I. P. SNOBOL, a string manipulation language. J. ACM. 11, 1 (Jan. 1964), pp. 21-30.

[FG] Feldmann, J. and Gries, D. Translator writing systems. **Comm. ACM** 11,2 (Feb. 1968), pp. 77-113.

[GR] Gerhart, S. L. **Verification of APL Programs**. Ph.D. Thesis, Carnegie-Mellon Univ., 1972.

[GE] Geschke, C. M. **Global Program Optimizations**, Ph.D. Thesis, Carnegie-Mellon Univ., 1972.

[GPP] Griswold, R. E., Poage, J. F., Polonsky, I. P. **The SNOBOL 4 Programming Language** (Prentice-Hall Inc. 1968).

[HA] Habermann, A. N. Synchronization of communicating processes. **Comm. ACM** 15, 3 (Mar. 1972), pp. 171-.

[HAN] Hansen, G. Work in progress at Carnegie-Mellon Univ.

[HE] Hewitt, C. Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot. AI Memo No. 251, MIT Project MAC (April 1972).

[HT] Hintz, R. G. and Tate, D. P. Control Data STAR-100 processor design. **IEEE CompCon**. (Sept. 1972), pp. 1-4.

[HO] Hoare, C. A. R. Notes on the theory and practice of data structuring. Working paper for 1968 Software Eng. Conf. in Rome.

[HO] Hoare, C. A. R. An axiomatic basis for computer programming. **Comm. ACM** 12, 10 (Oct. 1969), pp. 576-580 and 583.

[HO] Hoare, C. A. R. Procedures and parameters: an axiomatic approach. Symposium on Semantics of Algorithmic Languages. E. Engeler (ed.), (Springer-Verlag, 1971), pp. 102-116.

[HO] Hoare, C. A. R. Programm proving: jumps and functions. **ACTA Informatica** 1, 1972, pp. 214-224.

[HU] Hopcroft, J. E. and Ullman, J. D. **Formal Languages and their Relation to Automata** (Addison-Wesley, 1969), 81-84.

[HOP] Hopkins, M. A case for the goto. Proc. **ACM National Conference** (Aug. 1972).

[IA] Ianov, I. The logical schemes of algorithms. 1958 (English translation in: **Problems of Cybernetics** 1, Pergamon Press, 1960).

[IBM] **IBM System/360 PL/1 Reference Manual**, IBM Corp., C28-8201-0, Data Processing Devision, White Plains, N.Y. (1967).

[IV] Iverson, K. E. **A Programming Language** (J. Wiley, 1962).

[KI] King, J. **A Prog, am Verifier**, Ph. D. Thesis, Carnegie-Mellon Univ., 1969.

[KN] Knuth, D. E., **The Art of Computer Programming: Seminumerical Algorithms Vol** 2 (Addison-Wesley, Reading, Mass., 1969), pp. 398-400.

[KR] Krutar, R. a. Conversational systems programming by incremental extension of system configuration. **SIGPLAN Proc. of the Int. Symposium on Extensible Languages**, Grenoble, France (Sept. 1971), pp. 113-116.

[LR] Lombardi, L. A. and Raphael, B. LISP as the language for an incremental computer. Project MAC Memorandum MAC-M-142, MIT, (Mar, 1964).

[McC] McCarthy, J., et al. **LISP 1.5 Programmer's Manual** (MIT Press, 1965).

[McC] McCarthy, J. Recursive functions of symbolic expressions and their computation by machine. **Comm. ACM** 3 (Apr. 1960), pp. 184-195.

[McCR] McCracken, D. D. **A Guide to FORTRAN Programming** (J. Wiley, 1961).

[MI] Mitchell, J. G. **The Design and Construction of Flexible and Efficient Interactive Programming Systems**, Ph.D. Thesis, Carnegie-Mellon Univ., 1970.

[NA] Naur, Peter (ed.) Revised report on the algorithmic language Algol 60. **Comm. ACM** 6 (Jan 1963), pp. 1-17.

[NE] Newell, A. (ed.) **Information Processing Language-V** (Prentic-Hall, Englewood Cliffs, 1961), pp. 1-244. IPL-V.

[NI] Nilsson, N. J. **Problem Solving Methods in Artificial Intelligence** (McGraw-Hill, 1971) 116-155.

[NO] Noonan, R. E. **Computer Programming with a Dynamic Algebra**, Ph. D. Thesis, Purdue University, 1971.

[PA] Parnas, D. L. On the criteria to be used in decomposing systems into modules. **Comm. ACM.** 15, 12 (Dec. 1972), pp. 1053-1062.

[PAK] Pakin, Sandra. **APL\360** Reference Manual (Science Research Associates, 1968).

[RE] Reynolds, J. C. GEDANKEN--a simple typeless language based on the principal of completeness and the reference concept. **Comm. ACM.** 13, 5 (May 1970), pp. 308-319.

[RE] Reynolds, J. Definitional interpreters for higher order programming languages, Tech. Rep., Sys. Info. Sci. Dept., Syracuse Univ., 1972.

[RO] Rogers, H. **Theory of Recursive Functions and Effective Computability** (McGraw-Hill, New York, 1967), p. 67.

[RU] Rulifson, J. F. et al. QA4: a procedural calculus for intuitive reasoninng. SRI AI Center Technical Note 73 (Nov. 1973).

[SCH] Schumann (ed.) **SIGPLAN Proc. of the Int. Symposium on Extensible Languages** Grenoble, France (Sep. 1971).

[SC] Schwartz, J. Notes on the algorithmic language SETL, Courant Institute, New York Univ., N. y.

[SN] Snyder, L. **An Analysis of Parameter Evaluation for Recursive Procedures,** Ph.D. Thesis, Carnegie-Mellon Univ., 1973.

[vW] van Wijngaarden, A. Recursive definition of syntax and semantics. **Formal Language Description Languages for Computer Programming** (T. B. Steel, Jr., ed.) (North-Holland Pub. Co., Amsterdam, 1966), pp. 13-24.

[vW] van Wijngaarden, A., et al. Report on the algorithmic language Algol 68. **Numerische Mathematik** 14, 2, 1969, pp. 79-218.

[WA] Waite, W. M. A language independent macro processor. **Comm. ACM** 10, 7 (July 1967), pp. 433-440.

[WA] Waite, W. M. The mobile programming system: STAGE2. **Comm. ACM** 13, 7 (July 1970), pp. 415-421.

[WM] Wells, M. and Morris, James (ed.) **SIGPLAN Proc. of a Symposium on Two-Dimensional Man-Machine Communication** (Oct. 1972).

[WI] Wirth, N. The programming language Pascal. **ACTA Informatica.** 1, 1, 1971, pp. 35-63.

[WW] Wirth, N. and Weber, H. EULER--a generalization of ALGOL and its formal definition: Pt. I, Pt. II. **Comm. ACM.** 9, 1 and 2, (Jan. and Feb. 1966), pp. 13-25, 89-97.

[WG] Wile, D. S. and Geschke, C. M. An implementation base for efficient data structuring. **Int. J. of Comp. and Inf. Sciences** 1, 3 (Sept. 1972), p. 209-224.

[WU] Wulf, W. Programming without the goto. IFIP, 1971.

[WU] Wulf, W. et al. Bliss: a language for systems programming. **Comm. ACM** (Dec. 1971), pp. 780-790.

[WU] Wulf, W., et al. Bliss Reference Manual, Computer Science Department Report, Carnegie-Mellon Univ., 1970.

[WU] Wulf, W. et al. A case against the goto. Proc. ACM National Conference (Aug. 1972).