AN EXPRESSION VERIFIER FOR AN INTERACTIVE PROGRAM GENERATOR

By

JOHN FREDERIC LUCAS

Bachelor of Arts
Claremont McKenna College
Claremont, California
1975

Master of Arts
Claremont Graduate School
Claremont, California
1980

Submitted to the Faculty of the Graduate College
of the Oklahoma State University
in partial fulfillment of the requirements
for the Degree of
MASTER OF SCIENCE
December, 1982

AN EXPRESSION VERIFIER FOR AN INTERACTIVE PROGRAM GENERATOR

Thesis Approved:

R. E. Hedrick
Thesis Adviser

D. D. Fisher

M. J. Folk

Norman N. Durham
Dean of Graduate College

PREFACE

The aim of this study is to uncover and to resolve the fundamental issues facing the designer of an expression verifier system for an interactive program generator. The work presented here is intended to serve as the basis for an implementation of a verifier. It investigates such major questions as the division of the system into modules and the nature of the data structures used throughout the system. It also takes up, on occasion, the choice of algorithms to be used to accomplish certain system functions and the way in which these may be tailored to meet the special demands of this application. But it does not seek to provide a detailed description of the workings of each module in the system. In short, this study represents my conception of the design work needed to set the stage for an implementation.

I wish to thank my thesis adviser, Dr. G. E. Hedrick, for his guidance during this project and throughout my work at Oklahoma State Univerisity. I also thank Dr. D. D. Fisher and Dr. M. J. Folk for serving on the thesis committee.

Special thanks go to my friends John Warren and Arlen Long. Their concern and encouragement helped me through the frustrations and disappointments I faced from time to time as I worked on this project.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

BACKGROUND, MOTIVATION, AND METHOD

Introduction

An expression verifier is a set of routines which work together to determine whether a given text string constitutes a valid arithmetic, logical, or string-valued expression in some specified high-level programming language. The work presented here focuses on the challenges confronting the designer of an expression verifier which functions as a part of a program generator and which inspects candidate expressions supplied by the user of the generator. The processing of expressions to establish that they conform to the syntactic and semantic rules of a high-level programming language has been studied extensively in conjunction with the development of compilers for such languages. Program generators impose new and different requirements on expression verifying systems, and the techniques employed in compilers must be adapted to the new environment.

Later chapters describe the design problems posed by a specific expression verifier system and suggest solutions to them. The present chapter explores program generators and the contexts in which they require verification of expressions. It describes the hypothetical generator of the which the expression verifier discussed in the work is to be a part. It concludes by outlining the method to be used in raising

and resolving the design issues associated with the development of an expression verifier.

Program Generators: A Survey of Recent Research

A program generator is a computer program which writes other programs. More precisely, a program generator accepts as input a specification for a program and produces as output a program (typically a program in a high-level language) which meets that specification. The program generators currently in use or under development vary widely in the types of specifications they accept, but they are distinguished from other language processing systems by one common feature: they do not require the user to provide the program specification in the form of source text in some high-level procedural programming language.

Two related but distinct goals motivate the development of program generators. One is to make the power of the computer more widely available to users who are not trained in conventional programming techniques by allowing such users to specify their programs in a more natural way--that is, in terms more closely related to the way in which they conceive of the problem at hand and its solution--than is possible with an ordinary high-level programming language. The other goal is to make the process of software development less time-consuming and the end product more reliable by freeing programmers from some of the routine (yet highly detailed) work involved in constructing programs. The two goals are related in that to advance towards one is almost invariably to draw closer to the other. Yet they suggest somewhat different emphases in research. Attempts to achieve the first goal rely on techniques closely connected with the discipline of artificial intelligence,[1]

including methods for natural language processing and for automated problem solving. The second goal can be pursued through the adaptation and the extension of techniques originally devised in the development of compilers and other programming language processors.

Researchers who have in mind the first and more ambitious goal face two fundamental problems. They must devise a method for translating a user's "natural" specification for a program into some complete and consistent internal representation of this specification. They must then discover how to transform this internal specification into a program in the target language which meets the specification.

Heidorn [11] surveys four projects in which programs are generated based upon specifications obtained from the users through natural language dialogues. None of the projects reached definitive results, but each had some measure of success. The generators produced by each project worked within a rather restricted range of applications. Heidorn's own work, for example, involved generating programs for simple queuing simulations in a target language designed for simulation programs. Limiting the generator to a particular class of programs simplifies the processing of the natural language input since both the vocabulary of the specification and its range of possible interpretations are circumscribed.

The PSI project began at Stanford University and continuing at the Kestrel Institute, also aims at allowing the user to specify a program in a natural way [9, 10, 14]. In addition to natural language descriptions of the desired program, the user may supply example calculations and traces of the program's behavior. The system consists of a group of interacting modules (called "experts" in the jargon of artificial intelligence research[2]), each of which is responsible for some aspect of the

system's work. One carries on the dialogue with the user and helps to construct an internal representation of the program specification; another, the "coding expert", has responsibility for producing the target language program; still another, the "efficiency expert", attempts to perform optimization of the generated program. These experts are so called because, according to those who work in this field, they can be said in some sense to possess knowledge about some domain. It is appealing, but philosophically naive, to characterize PSI as a model of the human programmer who elicits a program specification from the end user and who then applies his or her knowledge of the problem domain, the fundamental techniques of programming, and the characteristics of the target language, to produce a suitable program.

Some researchers have chosen to set aside the problems associated with processing "natural" program specifications and have concentrated on the problem of transforming formal specifications into programs. Such formal specifications are cast in a form much less flexible than natural English. Yet they represent significant progress towards the goal of providing wider access to the power of computers in that they typically relieve the program specifier of the burden of choosing data representations and computational methods.

The task of transforming specifications into programs may be automated if two problems are solved. First, a set of mechanical transformations must be devised which, when applied in an appropriate sequence, will lead from the specification to a program. Second, a mechanical procedure must be developed to select a series of transformations which will produce the required program from a given specification.

Balzer [5] describes a system which ignores the second problem by requiring a human to select the appropriate transformation at each stage in the development of a program. The system's specification language provides for the description of a "world" (that is, a problem domain). The "world" includes the objects which populate it, the relationship which obtain among these objects, the constraints that objects must satisfy, the actions which apply to objects, and the rules of inference appropriate within the "world." The specification for an individual program within the problem domain includes a statement of the initial configuration of objects and a description either of the required final configuration or of the desired behavior of the system. The transformation of the specification aims at producing code which efficiently simulates the essentially nondeterministic process of discovering a sequence of actions which lead from the initial to the final state. Thus one important transformation "unfolds" the specified constraints on objects by introducing code which tests for violations of the constraints and enables the program to backtrack to a configuration at which an alternative series of actions may be attempted. The implementation of this system does not appear to be far advanced. Many of the transformations are left for the user to apply manually.

The work of Manna and Waldinger [15, 16, 17] has attempted to solve the problem of automatically selecting the transformation to be applied by adapting techniques from automated theorem-proving systems and program verifiers (that is, systems which determine whether a program is correct). Manna and Waldinger have implemented a system which produces programs for a restricted range of problems, those dealing with simple arithmetic and with list processing. Their approach is impressive in its theoretical rigor, and despite the relatively limited results to date, it seems

possible that their techniques may provide the basis for far more sophisticated systems.

Prywes [18] and others [14, 19] at the University of Pennsylvania have developed a program generator system which attempts to simplify the process of developing practical software (business file processing applications, for example) by permitting the user to specify a program by describing its data objects and the relations which obtain among them. Unlike the projects previously discussed, the University of Pennsylvania project (called MODEL II, for MOdule DEscription Language II) does not rely on artificial intelligence techniques; and while it produces genuinely useful programs, it requires a specification much more detailed--and much similar in appearance to an ordinary high-level language program--than the other systems. The generator works by organizing the user's specification into a directed graph describing the dependencies among the various data objects. By performing a topological sort on the graph, constructing input and assignment statements where these are implied by the specification, and constructing loops where the specification calls (implicitly) for iteration over a data aggregate (over all the records in a file, for example), the generator can produce code in a high-level procedural language (PL/I or COBOL). The designers of the system believe that a nonprocedural description of data objects and their relationships represents a more natural approach to instructing the computer for data processing applications than ordinary high level languages. The user need not be concerned with constructing input/output statements, nor about the ordering of program operations, nor even about the program's control structures. The program specification in the MODEL system is typically only about one-fourth as long as

the generated program; this suggests that development, debugging, and maintenance times will all be reduced substantially through the use of the program generator.

The Cornell Program Synthesizer developed for instructional use, makes no attempt to move beyond the high-level programming language as the mechanism for conveying instructions to a computer [24, 25]. The Synthesizer's user, in fact, still works directly with source text in a high-level language (PL/CS or PASCAL). Denning [8] describes the synthesizer as a "smart editor," that is, a system for entering or modifying programs which can make use of the syntax rules for the language to aid the user in manipulating the source code. The Synthesizer is designed for use in conjunction with a CRT terminal in an interactive environment. The user enters a program not by typing its text directly but by selecting templates for constructs in the language (procedures, blocks, loops, selection groups, and various classes of statements representing basic operations in the language) and filling in those details specific to his or her program. Each template is associated with a nonterminal symbol in the grammar defining the language and consists of text corresponding to sequences of terminal symbols and placeholders representing nonterminals. Constructing a program with the Synthesizer corresponds to deriving the program using the production rules for the grammar which defines the language. The system can thus ensure that at any stage in the entry of the program, the user's specification corresponds to a sentential form of the grammar. When the user has completely filled in all of the placeholders for all of the templates selected for the program, the program is assuredly syntactically correct. Certain very elementary entities in the language--identifiers, constants, and expressions--have

no templates. The user enters these directly. Whenever such an entry is made, it is inspected immediately to determine whether it is syntactically well-formed and appropriate for insertion at the point in the program where the user has typed it. The Synthesizer defends very thoroughly against the introduction of syntax errors into a program.

The Cornell approach admits of two variations. First, the templates for various target language constructs can be hidden from the user. The user of the Synthesizer places a template into the program using a special key sequence, then types additional text into the material displayed by the Synthesizer. In the variation suggested here, the user would select a construct from the menu and subsequently would be prompted for the information required to complete the construct. The user would not see the target language code. A second variation is to add "very high level" constructs, tailored to the range of applications for which the generator is intended, in addition to the primitive elements which have direct counterparts in the language. Roth [22] suggests, for example, that 50 to 90 percent of all business-oriented applications involve the relatively routine operations of formatted data entry, formatted data output, sorting, and simple file updates. Although the underlying code for any of these operations may be fairly complicated, a program generator system designed for business use could treat them as primitives. When the user elects to include one of them in a program, the generator responds by prompting for the information needed to tailor the operation to the specific application. Johnson [12] surveys a broad range of program generators currently on the market, many of which incorporate one or both of these variations.

The term "program generator" has been applied to a wide range of systems. The feature common to all of the systems described here is that they dispense with standard high-level language source text as the medium through which the user conveys instructions to the computer. Even in the Cornell Synthesizer, where the user manipulates the synthesized code directly, the fact that program entry is guided by the syntax rules of the target language represents a significant transformation in the way in which an ordinary high-level language is used to specify a program: the specification lies no longer in the source text as such but in the sequence of template selections through which the program is derived.

Expression Verification in Program Generators

Expressions--arithmetic expressions in particular--constitute a formal and artificial language for describing computations of various kinds. The notation is so convenient and so widely used, however, that it can be taken as a rather natural means of conveying some aspects of a program specification to a computer. No matter how sophisticated program generators become in their ability to process subsets of natural languages, it seems unlikely that they will dispense with standard infix notation as at least one alternative method for describing calculations. Thus some mechanism for establishing that an expression supplied by the user is syntactically well-formed and free of semantic defects is likely to be a part of any program generator system.

In systems employing sophisticated natural language processing techniques, however, verification of expressions will seem a trivial matter in comparison with the effort required to decipher the user's

less formal specifications. The problem is more interesting in the less complicated systems which can be fully implemented today. In generators such as the Cornell synthesizer and the commercially available, menu-driven systems, expressions represent the most complex input supplied by the user.

In these simpler program generators, expressions have a variety of uses. The most obvious, perhaps, is in describing the computation of a value to be assigned to some data object. Logical-valued expressions are an important component in control structures such as loops and selection groups. Indexed loops make direct use of numeric-valued expressions to specify the range of values to be assumed by the index variable. Expressions of any type are generally valid as items to be written in an output operation. Integer expressions are used to specify a record of a direct-access file in systems where a relative record address is permitted as a key. String expressions may play the role of keys in systems which support indexed files. In short, the user of a program generator like the Cornell Synthesizer or some menu-driven system will be required to supply expressions quite frequently.

The principal role of an expression verifier in such a generator is to determine whether a user-supplied candidate expression conforms to the rules for forming expressions imposed by the generator. In an interactive system, the expression must be examined immediately after the user enters it. If the expression is not valid, the generator must issue a diagnostic message and request a new entry. Among the more fundamental design issues is the question of how precise the diagnostic messages must be; the decision should reflect the designer's view of the needs of the intended user audience.

Aim and Method of this Study

The work presented here describes the requirements and proposes a design for an expression verifier for an interactive menu-driven program generator system. The objective is to provide a sound basis for the implementation of a verifier.

Ordinarily an effort such as this would take place in the context of a design project for a complete program generator. Such is not the case here. Although many elements of an expression verifier's operation depend upon the details of the organization and function of the system into which it is embedded, it is not essential to have in hand a detailed description of that system in order to take up the central issues in the design of the verifier. This study relies on a high-level description of the program generator of which the verifier is to be a part and leaves to the implementor the decisions which will depend upon the details of the program generator's design.

The hypothetical program generator in which the expression verifier described in the following chapters is to operate is a menu-driven system like those available commercially. It is to be used, typically, by persons in a small business environment to develop modest data processing and computational applications not supported by off-the-shelf packages. The user is not likely to be a professional programmer. The program generator, therefore, must be somewhat tutorial in character. The computer on which the generator executes is likely to be small, given that it is operating in a small business. Thus the designer of an expression verifier for this program generator faces two challenges:

1. In keeping with the orientation of the generator to the non-professional user, the verifier must provide clear, precise diagnostic messages when it is presented with an invalid expression.

2. In order to allow the program generator to run on a system with limited primary storage capacity, the verifier must be compact.

The major burden of the work presented here is to devise a verifier system which meets both of these requirements.

Chapter II presents the requirements which the verifier system must meet. Certain of these--those related to the hardware environment, for example--can be stated briefly and straightforwardly. Where the system's interaction with the user through its error-handling facilities is concerned, however, more detail is needed. The requirements must specify how the verifier is to respond to the wide range of potential errors. Thus much of Chapter II is devoted to an exploration of what it means to provide clear, precise diagnostic messages in this setting.

Chapter III addresses itself to the question of how the requirements imposed upon the verifier can be met. The goal is to produce a high-level system design. Such a design must specify the basic organization of the system--its division into modules and the apportionment of tasks among them. Further, the design must describe the paths of communication among the modules, including globally accessed data structures. Moreover, although the selection of algorithms to carry out system functions is often a detail best left to the designers or coders of the system's component modules, certain of these choices may be of such fundamental significance to the entire system that they are appropriately made as part of the system design. Such is the case here with the selection of a technique for parsing expressions, and considerable attention is given to this problem.

In an industrial setting, the results of a project such as this one would be communicated as tersely as possible. A simple enumeration of

the design decisions would be sufficient to permit module designers and coders to implement the system. Academics, by contrast, have the obligation--or are permitted the luxury--of providing a rationale for their design decisions. The present paper takes the academic attitude; accordingly, much of it represents an exploration and evaluation of alternative solutions to design problems.

ENDNOTES

[1]Raphael [21] offers a clear (if somewhat nontechnical) introduction to the major field of research in artificial intelligence. Weizenbaum [26] is a sharp critic of this research and argues that its accomplishments are frequently overstated.

[2]The term "expert," as it is used here, reflects the notion that the module embodies "knowledge" related to the task at hand. Thus PSI's coding module is an "expert" because it "knows" about fundamental programming techniques. The present study takes the terms "expert" and "knowledge" in a purely technical sense, with no implicit suggestion that the PSI system models human cognitive processes.

CHAPTER II

REQUIREMENTS FOR AN EXPRESSION VERIFIER

Introduction

The basic goals and assumptions which undergird this design project and which were described in the first chapter have important implications for the organization and operation of the expression verifier. The present chapter lays out--in detail--the standards of performance which the verifier must meet and the constraints within which it must operate.

These fundamental requirements for the verifier fall into four categories. First, the hardware environment within which the verifier will operate must be specified. Second, the nature of the expressions to be verified must be defined precisely. Third, the relationship between the verifier and other modules in the program generator system must be described: specifically, it is essential to take note of the nature of the information the verifier must supply to the routines which call it as well of the support which other components of the generator may provide for the verifier's operation. Finally, it is vital to specify in some detail how the verifier is to appear to the user of the program generator, particularly in the event that it detects an error in a user-supplied expression.

The focus of the present chapter is on the performance expected of the verifier and on the limitations within which it must operate. The question of how the verifier is to meet the goals established for it is

deferred until chapter three. The selection of the methods of implementation--the choice of algorithms and data structures to be used in the verifier--must wait until the design requirements for the system have been specified clearly.

The Hardware Environment

The hypothetical program generator of which the expression verifier is to be a part would operate on a microcomputer designed for use in a small business environment. These systems typically have somewhat more memory capacity than the less expensive "personal" microcomputers. They are disk-based systems--that is, one or more flexible disk drives are included as part of the minimum system configuration and the basic operating system provides for disk input/output operations and for maintenance of disk files. In contrast to the smaller "personal" machines, the basic system software is stored on disk and loaded into primary storage as needed rather than residing permanently in non-volatile read-only storage.

These microcomputer systems do not have a virtual memory facility. The microprocessors typically used as the central processing units in these machines have the capability of addressing only 64K bytes of storage, and the entire addressing space is generally occupied by physical memory. Substantial business application packages often cannot be fit into the available primary storage, so that some mechanism for loading program modules from disk during program execution is essential. Two such strategies have been developed. The more primitive, chaining, requires that the programmer include, at the end of a program segment, a statement specifying the disk file from which the next segment to be

executed is to be loaded. Upon execution of such a statement, the present segment is expelled from primary storage and the new segment takes its place. Conceptually, a "chain" operation is identical to an unconditional branch. A more sophisticated technique--one borrowed from early mainframe computers which had limited primary storage--involves the use of overlays [13]. In this scheme, the basic units transferred from disk to primary storage are the independently-compiled subprograms (external procedures) which make up the program. No special statements are required within the program to initiate a disk-to-primary storage transfer; any attempt to invoke a subprogram not in storage activates the overlay manager and causes the subprogram to be loaded. The burden for the programmer comes at the time the object modules are to be linked into an executable package. The programmer must assign each module a load address and must take care that no module is assigned to the same storage space as any of the modules it may call, directly or indirectly, lest it be overwritten by one of them. Neither chaining nor overlaying represents an ideal system for managing the memory hierarchy, but of the two the overlay mechanism is clearly preferable since it supports the division of a large program into callable subprograms. The chaining technique allows--in fact forces--a larger program to be broken into modules, but supports only the unconditional branch as a means for transferring control between modules.

An interactive program generator clearly will be a large program. The interactive "front end", which acquires the program specification from the user, will be large because it must support an extensive hierarchy of menus and provide for highly defensive error checking whenever it accepts user input. The expression verifier itself--which, as the

detailed design study will show, is a truncated compiler for expressions--will also be a sizeable module. It is reasonable to assume, for design purposes, that the entire program generator cannot be made to fit in the available primary storage. Moreover, since the expression verifier is a utility routine likely to be invoked by a large module already in storage, it is safest to suppose that the verifier itself may not be permitted to reside in storage as a single unit. It must be designed so that it may be loaded in several independent segments. The present design assumes that an overlay mechanism is available to support this division into modules.

The foregoing description of the hardware environment and of its implications for the organization of the verifier is probably sufficient to guide most of the design decisions. A more specific description is needed, however, to provide a standard against which the final product may be measured. For the purposes of this study, the target machine for the program generator of which the verifier is a part is a microcomputer with 64K bytes of primary storage and 500K bytes to 1000K bytes of secondary storage in the form of flexible disk drives. The system runs the somewhat dated but widely available operating system CP/M[1].

The Expression Language

The designer of an interactive program generator must choose between two distinct methods for providing the user with a language in which to express computations, string manipulations, and logical operations. One is to require the user to use the expression sublanguage of the programming language for which the generator produces its code. In this scheme, the expression supplied by the user in response to a request from the

generator is incorporated without modification into the generated code. The alternative is to make the generator's expression language essentially independent of the language to be generated and to require the verifier, as it is checking the validity of the user-supplied expression, to translate it into a form acceptable to the target language processor. For even greater flexibility, the verifier could translate a user expression into some intermediate form (a prefix or postfix notation, perhaps) which could later be translated into one of a variety of programming languages.

The second alternative is clearly the more complex, since it requires that the expression verifier include the code needed to effect a translation of the candidate expression. The approach has two potential benefits: first, it might provide the user with a more natural expression syntax, particularly if there are flaws in the design of the target language; and second, it supports the potential use of a single program specification in producing functionally equivalent programs in several different languages. A verifier which did not incorporate a translation mechanism would not contribute to producing such versatile program specifications.

The decision between the two approaches is relatively easy to make. Given that the typical user is a relatively unsophisticated programmer whose goal is to produce custom business applications, there is no real need for the program generator to be capable of producing programs in several different languages. If, moreover, the one language which is produced by the generator has a reasonably natural expression sublanguage, then there is little point in providing the generator with its own such language. Hence for this project, the simpler approach--that of using the target programming language's expression syntax--is taken.

The choice of the target language is a design decision which cannot be made within the confines of a discussion of the expression verifier. Many factors quite unrelated to the problem of verifying expressions--the suitability of the language for the applications likely to be required by the generator's users, for example, or the ease with which code can be generated--enter into the selection. Here the target language is accepted as a given for the design process. That language is CBASIC[2], a dialect of BASIC aimed at business applications. The commercial orientation of the language, with its relatively extensive file-handling capabilities, its PICTURE-like output formatting, and its 14-digit binary-coded floating point decimal arithmetic, is compatible with the applications likely to be developed by the typical user of the program generator.

CBASIC supports three data types: integers, real numbers, and variable-length character strings. Objects of type integer are completely interchangeable with those of type real; that is, the one may appear within an expression anywhere the other is syntactically valid. Expressions involving mixed numeric modes are valid. All of the familiar arithmetic operations--exponentiation, multiplication, division, addition, and subtraction--are included in the language. Logical operations are provided as well, but there is no special logical data type. Instead, the logical operations apply to numeric operands; the integer value -1 represents a logical "true", while all other values are taken to be logical "false". The concatenation operation is provided for character string data. Numerous built-in functions--including some for string manipulation--are provided.

CBASIC has no statement for declaring the type of an identifier. Instead, identifiers of type integer must be given names ending with the

type character "%" and those of type string must have names ending in "$". An identifier ending neither in "%" nor in "$" is taken to be of type real.

The language supports only one form of data aggregate, namely, the array, which may have up to three dimensions. Array space is allocated dynamically upon execution of DIM statements, which specify the extents of arrays.

No formal specification of the syntax of CBASIC expressions is provided in the documentation for the compiler, but the informal statement of rules for forming expressions is sufficiently precise that one may readily find a context-free grammar which describes the expression sublanguage [6]. One such grammar is listed in Appendix A.

The type declaration mechanism in CBASIC is awkward. The type characters appended to the ends of identifier are easily forgotten. Moreover, a misspelled identifier cannot be detected before execution of the program, and even then the exact nature of the error may be obscured. If the program generator were to deviate from the CBASIC conventions and introduce an explicit declaration mechanism for variables, this source of potential errors could be eliminated. For the purposes of this project, such a mechanism is assumed. Most probably a declaration module will be among the routines through which the user provides a program specification to the generator; in addition, the expression verifier may permit the user to make a declaration whenever it encounters an undeclared identifier within an expression. For consistency, the declaration of an array will include the specification of its extent; thus only one DIM statement will be generated for an array in a given program.

The Interface with other Components of the Program Generator

The expression verifer is a utility routine called by other components of the program generator system. All of the verifier's potential callers require at a minimum that the verifier indicate whether the string of text passed to it constitutes a valid CBASIC expression. Some callers, however, require additional information about the expression. The design of the verifier must take these demands into account.

Often the calling routine requires information about the type of an expression found by the verifier to be valid. Since objects of type integer and those of type real can be interchanged at will in CBASIC, it is not important that the verifier distinguish between integer expressions and real expressions. It must, however, have the ability to report to its caller whether the expression passed to it is a string expression or a numeric (integer or real) expression. In some instances, the calling routine will accept an expression of one of these types only. This means that the expression verifier should refrain from issuing an error message or permitting error correction when a candidate expression, though defective, is clearly of the wrong type. This in turn implies that the calling routine must specify for the verifier which of the two classes of expressions it will accept. Thus type information flows in two directions: the caller indicates which types are acceptable in the context in which the call to the verifier occurs, and the verifier reports which type of expression it has found.

In still other circumstances, the calling routine may impose even more stringent restrictions on the class of expression it will accept. Where the user is asked to supply a target for an assignment or input operation, for example, only those expressions which can be associated

with a specific storage location (that is, only scalar variables or elements of arrays) are appropriate. Occasionally only a constant or only an identifier constitute acceptable input to the program generator. The expression verifier must be so organized that callers which accept only a subset of valid CBASIC expressions can somehow specify this restriction. The alternatives for implementing this feature will be considered as part of the description of the design of the verifier in Chapter III.

The User Interface

The Nature of the User Interface

The expression verifier in an interactive program generator is invisible from the generator's user much of the time. The verifier is called by other components of the generator system--by the module which constructs arithmetic assignment statements, for instance--whenever the correctness of the user's response to a prompt for an expression must be confirmed. As long as no error is detected, the user is aware of the verifier's operation only insofar as it introduces a delay in the system's response. Only when an invalid construct is encountered does the potential for interaction with the user arise. Specifying the user interface thus amounts to describing how--from the user's point of view--the verifier is to respond to errors in expressions.

If the expression verifier's task were merely to discriminate between valid and illegal expressions, the user interface would be insubstantial. The verifier, on detecting any error, would abort its processing of the user's input and issue (or signal its caller to issue) a message indicating that the input did not constitute a legal

expression. The user would be required to determine the exact cause of the problem and to retype the entire expression in its corrected form in order to proceed with the construction of a program.

Such a simple user interface may be appropriate in a program generator designed for use by a professional programmer who is unlikely to make many errors and who is skilled at locating errors when he or she does make them. Even the professional, however, might have some difficulty in locating certain kinds of semantic errors (e.g., the use of an identifier previously declared to be an array as a scalar) and could benefit from more precise diagnostics.

In a program generator aimed at the relatively naive user, the simple user interface is clearly unacceptable. A message indicating only that the input did not constitute a valid expression is likely to frustrate the user and could lead to a series of unsuccessful attempts to correct the problem. This is a special danger with expressions where many errors (especially those of a semantic character) are not readily apparent, especially to the inexperienced eye.

Given its intended user audience, the program generator under consideration here requires an expression verifier whose response to errors is considerably more sophisticated than a simple indication that an error of indeterminate nature has been detected. It is reasonable, given the tutorial character of the program generator as a whole, to expect the expression verifier to provide relatively detailed diagnostic messages. Moreover, since one of the objectives of the program generator is to reduce the amount of effort expended in keying in text, some provision for error correction (other than retyping an entire expression) is desirable.

It is tempting to let the preceding paragraph stand as the statement of requirements for the expression verifier's user interface. But it is simply too imprecise. There are a number of major issues concealed in it--issues which ought to be addressed directly, before fundamental decisions are made about the structure of the verifier and about the techniques to be used in its implementation.

The remainder of this section is devoted to making the requirements for the user interface more specific. The discussion begins with a consideration of what constitutes a "relatively detailed diagnostic message" and a sketch of some of the implications of the requirement for precise errors messages for the design of the verifier. It then takes up the matters of providing facilities for repairing errors and handling multiple errors. The section concludes by developing guidelines for handling the various classes of errors which the verifier is likely to encounter.

The Nature of the Diagnostic Messages

The goal of error processing in the expression verifier is to enable the user to make a valid entry quickly and with a minimum of frustration. Given this aim, it is clear that any diagnostic message issued by the verifier should provide all the information necessary to understand the error and to effect a successful correction.

Much can be accomplished towards this end by careful wording of the messages. But this alone is not sufficient. The verifier's logic and data structures must be fashioned so that the appropriate information is gathered and preserved as the user's input is scanned. In particular, the verifier must be able to pinpoint the defective portion of the

expression and must classify the error in a way which is meaningful to the user. These requirements have very definite implications for the system design: they rule out, for example, the use of top-down parsing with backtracking and operator precedence parsing with precedence functions in the syntax analysis phase of the verifier. These methods, while they adequately discriminate between valid and invalid expressions, do not return enough information to produce adequate messages.

It is not enough, however, that the expression verifier collect detailed information related to an error. The diagnostic message, to be useful, must precisely describe the nature of the error in terms meaningful to the user. There is a problem, at least occasionally, of mapping from the error information obtained by the verifier (usually encoded in the configuration of a recognizer for a regular or context-free language) into an error classification which seems natural to the user. Although this mapping may often be accomplished through a careful phrasing of the error message, it is sometimes necessary to tailor the verifier--for example, by writing the expression grammar in a way which permits the syntax analyzer to discriminate more finely between classes of errors--in order to obtain reasonable diagnostics.

In some cases it may be impossible or impractical to issue an accurate and precise message, perhaps because of some quirk in the syntactic features of the expression sublanguage or (more likely) because of some implementation constraint. Here it is probably best for the designer to be honest and to specify a message which states simply that an invalid expression has been encountered. Such a diagnostic is probably more helpful than a bewildering or potentially misleading one. The user

of an interactive system is conditioned to respond quickly and without a great deal of thought to messages from the system, and a diagnostic which admits of a misinterpretation is likely to lead the user astray.

Error Repair

Certain compilers--the Cornell PL/C compiler, for example--attempt to correct errors in a source program automatically, so that the program can proceed as far as possible towards successful compilation and execution [7]. The techniques which make this possible could be applied in the expression verifier, but this is not desirable. To do so would increase the complexity and size of the verifier. More importantly, automatic error correction is inappropriate in this environment, since any method which can accomplish it relies on some sort of assumption about the user's intent. Such assumptions easily can be incorrect. Since the user of an interactive program generator is available to make his or her intent clear, there is scarcely any point in guessing at it.

A more intriguing approach to handling errors in the expression verifier is to allow the user to correct a defective expression without requiring him or her to retype the entire expression. If, for example, the expression is missing a parenthesis, the user might be permitted to insert one at any position in the expression where it is appropriate. There are two advantages to this approach. First, it pinpoints the error and provides detailed information about how it may be corrected. Second, it eliminates the need for retyping the expression, a process which can be time consuming and may introduce new errors.

Unfortunately, this approach has a serious drawback. It is likely to add considerable complexity to the verifier. In analyzing an

expression, the verifier must isolate the token or tokens which give rise to the error and determine what might be put in their place in order to correct the expression. Often there will be several alternatives, some of which may be eliminated as more of the expression is scanned. With suitably sophisticated data structures and perhaps some clever augmentation of the expression grammar's production rules--not to mention a computer with ample primary storage--the task is feasible.[3]

One class of errors can be handled with a repair scheme, even in a microcomputer-based system. These are errors caused by the occurrence of undeclared identifiers in an expression. It seems eminently reasonable for the verifer to prompt for the needed attributes (type and, for arrays, bounds) on encountering an undeclared variable. Ideally the verifier would make use of the information it develops during the analysis of the candidate expression so that is can present the user with only those options for attribute values which are valid given the context in which the identifier appears. If an undeclared identifier is used in a context where only a scalar variable of numeric type is valid, for example, the verifier should allow the user to choose to make the variable of type integer or to make it of type real. The string type ought not to appear as an option on the variable declaration menu which the verifier presents to the user.

Multiple Errors

A candidate expression supplied by the user might contain several errors. The expression verifier can embody one of three distinct approaches to handling multiple errors:

1. It might signal the first error it encounters and halt.

2. It might incorporate error recovery (not correction) routines which permit it to continue analyzing the candidate expression after detecting an error. The verifier would save the information relevant to each error it found. Following a complete scan of the expression, the verifier would issue a diagnostic message describing all of the errors in the candidate expression.

3. It might attempt to detect and signal the error which--on some criterion or another--is most worthy of being brought to the user's attention. The verifier would incorporate error recovery routines so that it could scan beyond the first error.

The first alternative has the distinct advantages of being relatively easier to implement and of requiring less code (and hence less memory space). It relies on the assumption either that the order in which errors are signalled to the user is unimportant or that the order in which the verifier detects errors corresponds exactly to the most helpful order in which errors can be announced. This assumption is difficult to justify. Still, space constraints may dictate that this approach be taken. In such a case, it may be possible to organize the verifier so that it detects errors in an order which will make sense to the user; the goal would be to mimic, insofar as it is possible, the behavior of a verifier based on the third approach.

The second approach is considerably more attractive from the user's point of view than the first. Most importantly, the user has all of the information necessary to effect a complete correction when he or she re-enters an expression. There are two drawbacks, however: first, the user may be overwhelmed by a list of several errors; and second, this scheme makes it awkward to allow for user repair of a defective expression short of re-entering it.[4] One can argue, with some justification, that this sort of approach is not sufficiently interactive: it requires the user to respond to several not necessarily related pieces of error information all at once. In this respect, it has something of the

flavor of a batch-mode compiler. Still, this approach might be entirely appropriate in a program generator whose typical user is to be an experienced programmer.

The third approach focuses the user's attention on a single problem, thus avoiding the danger of discouraging him or her by presenting a long list of errors. The major disadvantage here, of course, is that error correction becomes an iterative process. Correcting an expression may require several attempts even if the user carefully follows the guidance given by the diagnostic messages. This could be extremely frustrating, although it is not clear that the relatively naive user would find this more discouraging than a scheme which lists all of the errors at once. The idea of leading the user step-by-step from an invalid expression to a valid one is consistent with the tutorial character of the program generator.

The present system will adopt the third approach, recognizing that it is not the only choice nor even the obvious one. An empirical comparison of the three strategies--testing user performance and user preference--might be in order here.

Selection of the third scheme for handling multiple errors brings with it the added burden of determining the order in which error messages are to be issued. Two ordering principles seem plausible:

1. The most discouraging error--that is, the one most likely to cause the user to abandon the candidate expression entirely instead of attempting to correct it--should be signalled first. The idea here is to reduce the wasted effort associated with making minor corrections only to be forced to give up entirely when a more serious problem is revealed.

2. The most subtle error--the one least likely to catch the user's eye--should be signalled first. The aim, once again, is to reduce wasted effort, but the view here is that the obscurity, rather than the severity, of an error should determine its position in the ordering.

The principles may often give the same result, but they need not do so. Thus the expression verifier's multiple error handling must reflect a choice between these.

Consider the expression

AZ(5) */ V,

where AZ and V both have been declared as scalar numeric variables. The sequence "*/" is a gross violation of the syntax rules of the language. The use of the subscript list following AZ is not so blatant an error: it could be correct, if only AZ were declared to be a one-dimensional array. Rule 1 gives little guidance here: both errors are relatively "discouraging". The user will have to replace AZ with a different identifier (or omit the subscript list) and choose a single operator in place of "*/". Rule 2 is considerably more helpful. The operator sequence "*/" is by its nature more likely to be noticed by someone who has any idea at all about how to form a valid expression than the extraneous subscript list on AZ. Thus rule 2 requires that the latter error be signalled.

The foregoing example illustrates the major difficulty with rule 1: it is very difficult to decide which errors are "discouraging" and which are not. Thus while the rule has a certain intuitive appeal, it is not especially attractive as a concrete guideline for ordering error messages.

Rule 2 is more useful in practice. This is largely because the various sorts of errors break fairly naturally into classes based upon their visibility. The simplest and most formal division is bipartite: there are errors which are purely syntactic (that is, they can be seen to be errors without regard to the attributes of the identifiers used in the expression), such as the operator sequence "*/"; and there are those

which have a semantic component, such as the problem with "AZ(5)" in the example just given.

This formal division is not quite adequate in practice. Not all purely syntactic errors are readily visible: the operator sequence "+-" (as in "I + -5") is formally the same sort of error as "*/", but because in ordinary arithmetic and algebra this sequence is meaningful, the error in "I + -5" is likely to be more obscure to the typical user. For practical purposes, errors may be classified into those which are not obscure (gross violations of syntax rules), such as the operator sequence "*/", those which are somewhat obscure (typically less obvious violations of syntax rules), and those which are obscure (typically errors with a semantic component). No strict formal criterion can be given for assigning a specific error into one of these classes; once again, an empirical test with members of the intended user community might provide the soundest basis for making the categorization. A serviceable classification based upon common sense is possible, however, and such will be the approach taken here.

In a system where the user is permitted to repair certain types of errors without re-entering the entire expression, rule 2 requires some modification. If repair of an error will have permanent consequences for the program--as, for example, the declaration of a previously undeclared identifier would--the error should not be signalled until the expression is correct in all other respects. Thus in the candidate expression

A - (X + S,

where X is a numeric scalar variable, S is a string variable, and A is not declared, it might be best to defer signalling that A is not declared

and accepting a declaration for it until after the missing parenthesis and type mismatch problems are resolved. If the user abandons the expression entirely because of one of these latter errors, there will be no spurious declaration of A because the user will not have been presented with the opportunity to make it. This is admittedly a conservative strategy: it is not all that likely that a user will declare a variable in an expression which he or she ultimately will discard. But there is little, if anything, to be gained by not providing this extra measure of protection for the user.

Practical Guidelines for Error Handling

The discussion of the requirements for the expression verifier's user interface has thus far proceeded at a general level. The present section considers how specific classes of errors are to be processed by the verifier.

The task of classifying the various potential errors in expressions is a difficult one. For the purpose of specifying the operation of the expression verifier as the user will see it, one needs a classification which corresponds to the user's own taxonomy of errors. The goal is to treat alike those errors which the user perceives as being alike. Unfortunately, there is no precise, formal classification of the ways in which humans perceive errors in expressions; hence a completely rigorous specification of the expression verifier's response to errors written from the point of view of the user is impossible.

An alternative approach to classifying errors ignores the user's perspective entirely. Instead, it considers the various ways in which the expression verifier might detect an error. To each such possibility

there corresponds a class of potential errors. This approach has the advantage of completeness: if the verifier accepts all and only valid expressions, and if one takes account of all the mechanisms by which a candidate expression may be rejected, then one can obtain a classification which encompasses every possible error. Establishing a taxonomy of errors in this way, however, requires that the expression verifier already exist, or at least that there be a very detailed design for it. If such were the case, there would scarcely be any point in specifying requirements for it! Moreover, to ignore the user's point of view in designing the user interface and to allow the characteristics of this interface to be dictated by the details of the implementation run counter to the understanding of interactive system design which informs this project.

The foregoing conclusions, as negative as they are, do not rule out the possibility of developing specifications for error handling which are sensitive to the user's viewpoint and relatively independent of implementation details. So long as one is willing to accept a tentative and perhaps incomplete categorization of errors, one can make considerable headway towards defining the expression verifier's responses. Given such an informal classification, one can consider each group in turn, determining exactly what information the expression verifier should report to the user and what priority errors of the group in question should be assigned in the scheme for handling multiple errors.

The scheme employed here divides potential errors into four classes. First, there are those which involve a missing or extra element (operator, operand, parenthesis, or comma) in the candidate expression. Second, there are defects within the individual elements of the expression--an

identifier containing an illegal character, for example, or perhaps a numeric constant which lies beyond the allowed range of values. Third, there are problems associated with the attributes of identifiers. These include errors such as mixed string and numeric modes, incorrect number or types of arguments for a built-in function, and defective or missing subscript lists for arrays. Finally, there are errors raised by undeclared identifiers, often the result of a misspelled name.

This classification is admittedly informal. Moreover, it is somewhat fluid: a given error may fit more than one category. Consider, for example, the candidate expression

X LR Y

If the user intended to type "X LE Y", then the problem with this expression is of the second class; the expression contains a defective relational operator. A less likely possibility is that LR is to be an identifier. In this case, the expression is missing two operators. Given some knowledge about the kinds of errors humans are likely to make, one can usually make an adequate guess at the correct classification of any given defect. Unfortunately it is often impractical to incorporate this ability into a computer program which is compact and rapid. Thus it is often the case that a program such as an expression verifier will make the incorrect classification when confronted with a defect that may be interpreted in more than one way. Without large and sophisticated algorithms for determining the "nearest" valid expression, for instance, an expression verifier will treat "X LR Y" exactly as it would treat "X AB Y", namely, as an expression with missing operators. Still, the classification suggested here is not so vague as to be useless.

For the first class of errors--those involving missing or extra elements--the expression verifier should record the exact position within the candidate expression at which the problem occurs. Where there is an extra element, it should be displayed for the user; where an element is missing, the verifier should supply a list of those elements which may legally appear at the point of the error. Errors involving missing elements are typically among the most obvious defects in an expression; hence such errors are of lowest priority when there are several to be announced to the user.

There are two exceptional cases within the class. First, an expression may contain an unequal number of opening and closing parentheses. In general it is impossible to determine whether the imbalance is due to a missing parenthesis or to a superfluous one. Moreover, it usually is not possible to state precisely where a parenthesis should be inserted (or which parenthesis should be deleted). Thus the expression verifier can do no more than to announce that a candidate expression is incorrectly parenthesized. Since this type of error can be rather subtle when the candidate expression is heavily parenthesized, it seems appropriate to assign it an intermediate position in the hierarchy of priorities for handling multiple errors. The foregoing considerations, it should be noted, apply to parentheses which enclose an expression or subexpression and not to those which set off a subscript or argument list. Problems involving such parentheses are to be treated according to the standard pattern for errors of the first class.

The second exceptional case among the first class of errors is related to CBASIC's treatment of the unary operators + and -. As in Fortran, these unary operators are equal in precedence to the binary

operators denoted by the same symbols. Thus the expression A + -5, although it is meaningful in ordinary algebra, is illegal in Fortran and in CBASIC. One can argue that this is a flaw in the design of these languages, and one can reasonably suppose that errors of this sort are likely to be more common and less obvious to the user than other cases involving extra operators or missing operands. Therefore, the expression verifier should give special treatment to errors caused by following a binary arithmetic operator by a sign. The diagnostic message should note the location of the illegal sequence and remind the user of the language's restrictions. In case of multiple errors, this type of defect should have intermediate priority for its announcement.

Defects within the individual elements of an expression--lexical errors, in the jargon of compiler writers--constitute the second major class of errors. These errors take on two principal forms: first, there are those involving an illegal character within an element (e.g., the use of an underbar (_) in a variable name), and second, there are those where a sequence of valid characters fails to constitute a valid element on other grounds (a numeric constant which is out of range, for example). In the former case, the expression verifier should isolate the problem by noting both the invalid character and the context in which it appears. Thus where the user places an underbar in what would otherwise be a valid variable name, the verifier should not merely bring the underbar to the user's attention but indicate that it is invalid as part of a variable name.[5] Errors involving illegal characters are difficult to place in the hierarchy for handling multiple errors. They would seem to constitute gross violations of the rules for forming expressions and thus to be relatively obvious. On this basis they

should be assigned a low priority. The example cited here, however, indicates that under some circumstances an error caused by an illegal character could be rather subtle. On the whole, however, such errors merit a low priority in the hierarchy for announcing multiple errors. The second group of lexical errors--those which do not involve an illegal character--are clearly somewhat more obscure and should be given an intermediate priority. In case of such an error, the verifier should bring the entire invalid element to the user's attention and provide a precise statement of the problem (e.g., "32768 is too large to be a valid integer contant").

The third class of errors--those associated with the attributes of identifiers (semantic errors, in the technical jargon)--is perhaps the broadest of the four. The feature common to all such errors is that they are not at all apparent unless one is aware of the attributes of the variables in the defective expression. Hence these errors have the highest priority for announcement when a candidate expression contains more than one error. The verifier should isolate the entire defect in the case of such an error. In the following example, let string.variable be a scalar of type string and x a scalar of type real:

```
string.variable + "five" ge x + 5 or x le 0
```

The expression is defective because the comparison is between subexpressions of different types. The diagnostic message issued by the verifier should be couched in terms of the subexpressions and not individual variables. A message which complained that x was of inappropriate type, for example, would not isolate the entire defect.[6]

Errors caused by the appearance within a candidate expression of an undeclared identifier--errors of the fourth class--represent the only

defects which the user may repair without re-entering the entire expression. Typically this kind of error comes about either because the user has misspelled the name of a variable previously declared or because the user failed to declare the variable in advance. This suggests that, ideally, the verifier should allow the user either to replace the name with that of a variable which has already been declared or to provide a declaration--on the fly--for the name. The options to abandon the entire expression or to obtain a listing of all variables declared so far should also be provided.

This approach to handling undeclared identifiers, though it is ideal in the flexibility it affords, may be too complex to be comfortable for the user. The user enters an expression in the context of specifying some aspect of a program to be produced by the program generator and his or her attention is likely to be focussed on this task rather than on the more narrow problem of supplying an expression. Under the ideal scheme, the appearance of an undeclared identifier in an otherwise acceptable expression sets into motion a lengthy and perhaps overly distracting sequence: (1) the verifier issues a diagnostic message and presents the options (abandon this candidate expression and make a new entry, replace the name, provide a declaration for the name, or defer the decision and examine a list of already declared variables); (2) the user makes a selection; and (3) if the user has not abandoned the candidate expression, the verifier initiates the processing for the selected option. The repair leads the user away from the real task of providing information from which a program will be constructed and is potentially confusing. The repair effort delays closure--the completion of the pending task--and such delays have been observed to impair the usability of interactive systems [23].

One way to reduce the complexity of the repair scheme is by eliminating some of the options. If the user is allowed to repair the expression only by replacing the undeclared name or only by providing a declaration of the identifier which appears in the expression, then one layer of choices may be removed from the system.[7] The option of replacing the name is clearly the one to be given up. If the user has simply misspelled the name, he or she may correct the problem by re-entering the entire expression with the proper spelling. If, on the other hand, the user forgot the declaration, his or her only recourse--in the absence of the repair mechanism--is to abandon the present task completely, move to the variable declaration module in the program generator, provide a declaration, and then return to the point at which the error occurred to re-enter the expression. Thus while the replacement option is a convenience, the declaration option is virtually a necessity.

The decision about restricting the repair options would best be made on the basis of an empirical test aimed at determining whether the flexibility of the ideal scheme make up for its complexity. Such a test, however, is a luxury which requires that both approaches be implemented. For the purposes of the present project, the single repair option--declaration of an identifier previously undeclared--is adopted, not only because it reduces the complexity of the user interface but also because it make less demand on storage space than the ideal scheme.

Errors involving undeclared identifiers require a special position in the hierarchy for announcing errors when a candidate expression contains more than one. As the discussion of multiple error handling has already noted, the repair process allows for the declaration of a new identifier. If such a declaration is made while there are other

errors in the expression, there is the risk that the user will later abandon the expression. In such an event, the disposition of the newly-declared identifier poses something of a problem: should the declaration by undone or left intact? The problem is best solved by not announcing an undeclared identifier until all other errors have been resolved.[8]

The foregoing guidelines for error processing are largely independent of the details of the implementation. They leave the designer or implementor of an expression verifier with the task of organizing the system so that it can support the error handling specified here. This effort has two distinct aspects. First, the detailed design of the system must be undertaken with the requirements stated here firmly in mind. The selection of data structures and algorithms will be influenced, at least to some degree, by the demands which are implicit in the specifications for error handling. Second, once the fundamental design choices have been made, the designer or implementor must map each error state of the verifier--each point at which an error may be detected--into one of the four classes described here. This mapping permits the designer or implementor to determine exactly what processing is needed to support the error handling called for by the requirements.

ENDNOTES

[1]CP/M is a trademark of Digital Research, Inc.

[2]CBASIC is a trademark of Compiler Systems, Inc.

[3]The notion of allowing the user to repair a defective candidate expression by modifying some isolated portion of it has a straightforward conceptual basis, even if the implementation is complicated. If the syntax analyzer constructs a representation for the parse tree for the candidate expression and supplies nodes with special "placeholder" symbols at those points where the user's input cannot be parsed, the user can then repair the expression by editing the parse tree. The placeholder symbols could indicate exactly what the user could put into the corresponding segment of the candidate expression in order to obtain a valid expression. The information encoded by the placeholders could be used to guide the user in making the necessary corrections.

[4]The problem lies in presenting the user with the opportunity to make such a repair. If only one error is reported at a time, a user-repairable error is signalled with a diagnostic message and a repair menu. If the verifier reports all errors in a candidate expression at once, it must provide for the case where there are several repairable errors. This is it can do either by presenting the user with a menu from which to select the error to be repaired next or by leading the user through the repairable errors in some predetermined sequence. Both alternatives are awkward.

[5]On a system in which the user/machine interaction makes use of the full screen of a CRT display, the verifier might, on encountering the defective identifier "customer_name", highlight the underbar character (_) and issue the message "'_' not permitted in a variable name." Where full-screen interaction is not supported (so that the verifier does not have access to the portion of the screen in which the user typed the candidate expression), the verifier's response might be "'customer_name' is not a valid variable name because it contains '_'." In both cases the diagnostic message might also include a positive statement of the rule for forming variable names.

[6]The ideal message for this example would indicate that the subexpression string.variable + "five" ge x + 5 illegally compares a string expression with a numeric expression. To support the production of such a message, the verifier would need to isolate this entire subexpression and determine that it was invalid.

[7]If the only repair option were replacement, for example, then the verifier could issue the diagnostic and ask immediately for the new

name. The user could signal that he or she wished to abandon a candidate expression by responding with a null entry. A similar scheme would work if the only option were to provide a declaration.

8This is not a full solution of the problem, since a candidate expression may contain more than one undeclared identifier. The user may provide a declaration for the first but abandon the expression when a subsequent undeclared identifier is announced. The policy adopted here is that any declaration once made is permanent.

CHAPTER III

A DESIGN FOR AN EXPRESSION VERIFIER

Introduction

The statement of requirements in Chapter II does not provide an adequate basis for implementing an expression verifier for an interactive program generator. While it describes in detail the performance expected of the verifier, it does not suggest how this performance may be achieved. The present chapter fills this gap.

A system such as an expression verifier poses two kinds of design issues. First, there are structural questions. These have to do with the division of the system into modules. Second, there are questions regarding the techniques to be used within the individual modules to achieve the required results.

The chapter begins with a consideration of structural issues. The major tasks which must be performed by the verifier system are identified. Then these tasks are apportioned among modules and a scheme for placing them in primary storage is developed. The discussion of structural questions ends by considering how the organization of the verifier and its interface with other program generator modules is influenced by the information requirements of its callers.

The exploration of techniques which might be used in the verifier dominates the remaining pages of the chapter. The syntax analysis module--the central component of the verifier system--poses the most

challenging questions for the designer and receives the most careful attention. The aim of the discussion of the syntax analyzer and the other modules is not to provide a minutely detailed description of their inner workings. Rather, the goal is to uncover and to resolve in a consistent and considered manner the fundamental problems likely to be encountered in developing a verifier system which meets the requirements established in Chapter II.

The Structure of the Expression Verifier

Functional Anatomy of the Verifier

The expression verifier can be viewed as kind of truncated compiler. It analyzes expressions for correctness without generating code to carry out the operations specified in them. Thus the verifier will bear considerable resemblance in structure to a compiler.

A typical approach to analyzing source code in a compiler divides the processing into a lexical phase and a syntactic phase. Although the division of labor between the lexical analyzer and the syntax analyzer varies from one system to another, the usual pattern is to make the lexical analysis routine responsible for grouping the individual characters of the source text into the basic entities of the language: keywords, operator symbols, constants, identifiers, and the like. These entities (or "tokens") typically can be described formally using regular grammars and therefore can be recognized by finite-state automata. Syntax analysis proceeds on a token-by-token basis; its goal is to reconstruct a derivation of the token string representing the source code based upon the grammar specifying the source language. The division between lexical and syntax analysis is largely a practical one; it is

more efficient (in terms of execution time and perhaps also of space) to employ a separate recognizer for the basic entities of the language than it is to require the syntax analyzer to work at the character level and to include descriptions of the tokens in the grammar which guides syntax analysis [1].

This division of labor is entirely appropriate in an expression verifier. The verifier's lexical analyzer must recognize the basic entities of expressions, namely, identifiers, constants, and operators. These all can be described formally with regular grammars; the lexical analyzer thus may consist of a module which simulates the actions of a finite state automaton (or of a group of such modules). The verifier's syntax analyzer must recognize the context-free language consisting of all valid CBASIC expressions. It will be based upon some well-known parsing method.

Compilers typically record information about identifiers used in the source code into a symbol table. The program generator of which the expression verifier under consideration here is a part maintains such a table. It stores CBASIC reserved words, the names of CBASIC built-in functions and a coded representation of the number and types of arguments taken by each, logical file names, labels associated with program segments, and the names and attributes of user-declared variables. The expression verifier will have occasion to look up information in the symbol table and to insert information. The design of the verifier presented here assumes the existence of routines which support both kinds of access to the symbol table. Since the table is used by the program generator in a number of contexts unrelated to the processing of expressions, and since its organization is strongly influenced by these

other uses, a detailed description of the data structures and the manipulation routines involved is not appropriate here. It suffices to insist that the verifier have ready access to information about any identifier it may encounter in an expression.

The lexical and syntax analyzers, in conjunction with the symbol table access routines, perform the fundamental tasks associated with verifying the correctness of expressions. There are other tasks to be performed, however, particularly with respect to error handling. The requirements described in Chapter II make it clear that no verifier routine may simply issue a message and halt upon encountering an error. In order to support the error handling specified by the requirements, the verifier must maintain a record storing the essential information about the error with the highest priority for announcement which it has encountered at the current stage of its processing. When an error is encountered, the routine which detects it must update this record if the new error has a higher priority for announcement. At the conclusion of all processing by the lexical and syntax analyzers, an error message display routine issues a message based upon the contents of the error record.

In addition to producing error messages, the verifier must provide for convenient error recovery in the event that the candidate expression contains an undeclared identifier. Thus a variable declaration module will be part of the verifier system.

The foregoing represents a division of the expression verifier into subcomponents based upon the tasks which the verifier must perform to achieve its goal. This division is largely independent of any particular implementation. It does not give a detailed description of the system

organization needed to fulfill all of the requirements set out in Chapter II. Producing such a description involves a more careful scrutiny of those requirements and their implications.

Organizational Constraints Imposed by the Hardware Environment

The requirements which most profoundly affect the organization of the verifier are those related to the hardware on which the program generator is to run. The target machine is relatively small; it has only 64K bytes of primary storage. The expression verifier system, being a utility routine called by other components of the program generator system, may occupy only a fraction of this space. In fact the entire verifier cannot be resident in primary storage all at once, even if it relinquishes its space when it is no longer needed. The verifier must be structured so that it may be brought into storage by parts.

If the target machine supported a virtual storage scheme, the task of breaking the verifier into pieces could be left to the hardware and the system software. In fact, only a static overlay mechanism is available. This system requires that the programmer determine in advance how primary storage is to be allocated at run-time. The principal method for using the overlay scheme effectively involves breaking a system into modules--callable procedures--which do not call each other and which require approximately the same amount of storage at run-time. Such a group of modules can be assigned to the same region of primary storage; the overlay manager--part of the system software--has the responsibility for loading individual modules from secondary storage as each is needed.

One approach to organizing a large utility program in order to exploit the overlay scheme to advantage involves dividing the necessary

processing into a sequence of consecutive steps and constructing the major modules of the utility so that they correspond to these steps. These modules all share the same space in primary storage. A small controlling module, permanently resident in primary storage and responsible for the interface with routines which call the utility, calls each of the major modules of the utility in turn. Each call from the controlling routine starts a new phase of processing and brings a new module into the region of primary storage set aside for the utility.

The principal advantage of such a scheme is that it permits large utility routines to be included in a system without demanding a correspondingly large portion of primary storage. Moreover, it is somewhat flexible. Typically there are several alternatives for dividing a utility's operation into phases; one can often trade an increased number of modules (which implies some increase in the size of the controlling module) for a smaller region of reserved space in primary storage. Further savings of primary storage are achieved when several independent large utilities are broken up in this way: one can assign all of the major components for <u>all</u> such utilities to the same storage space.

The scheme has its price, however. Moving code from secondary storage to main storage takes time; when the secondary storage device is a flexible disk drive, the time is not insubstantial. In an interactive system such as a program generator, response time is at a premium and delays introduced as pieces of a utility routine are loaded can be annoying to the user. Unfortunately, there is little choice but to pay the price. A useful program generator is a sophisticated system and is necessarily large; if it is to execute on a small system, it must rely heavily on the use of overlays. Response time will suffer.[1] In order

to minimize this ill effect where large utility routines are concerned, the number of modules which share storage space--and consequently the number of loads from disk required to complete the processing--should be kept small.

Despite the response time problem, this scheme is the method of choice--rather, of necessity--for the expression verifier. The principal question is that of how to divide the system into modules which can be called into primary storage in succession.

Perhaps the most obvious division is along the lines of the basic functions outlined earlier: lexical analysis would come first, followed by syntax analysis, followed, if necessary, by error message production and error repair. This represents a natural separation of the expression verifier's processing; by contrast, for example, one would not wish to split syntax analysis into several parts because of the large amount of information which would have to be passed from one part to the next. Indeed it is difficult to see how any other division could work.

Adopting this division has some significant--and not altogether benign--consequences for the workings of the verifier system. In particular, it calls for the complete independence of the lexical and the syntax analyzers. The candidate expression is processed by the lexical analyzer and reduced to a string of tokens. The syntax analyzer then takes the place of the lexical analyzer in primary storage and processes the token string. An alternative approach employed in many compilers makes the syntax and lexical analyzers co-resident in primary storage. Lexical analysis is performed on a token-by-token basis at the request of the syntax analyzer; that is, the syntax analyzer calls the lexical analyzer whenever it needs the next token in the input stream.[2]

The difference in organization is inconsequential so long as the only goal of the processing is to determine whether the input string belongs to the language recognized by the system. When precise diagnostics are required in the event of an error, however, the scheme which makes lexical analysis subordinate to syntax analysis has some advantages--advantages which the expression verifier under consideration may be forced to forego because of space constraints.

First, the method which has the lexical analyzer reduce the input string to a string of tokens all at once puts some distance between the user's input and the syntax analyzer. Syntax analysis in this case works with the token string; the input string is not relevant to the processing which is taking place. Any error detected in syntax analysis is a defect in the token string. Error messages must make reference not to the token string, however, but to the text supplied by the user. Where lexical analysis is subordinate to syntax analysis, the syntax analyzer is working with the input string, albeit somewhat indirectly. In a system so organized, a syntax error is detected initially as the appearance of an inappropriate token, but since the input string is still present and has not been processed beyond the point at which the error occurred, it is relatively easy to issue a diagnostic which refers to the text supplied by the user.

Second, the relationship between the lexical analyzer and the syntax analyzer influences the way in which lexical errors may be handled. Where the lexical analyzer must pass through the entire input string and reduce it to a string of tokens all at once, its processing is guided purely by the text; it has no information about the syntactic structure of the string. By contrast, where the lexical analyzer is called on a

token-by-token basis by the syntax analyzer, it has access--at least potentially--to information about what kind of entity legally may appear next in the input string. This information is not particularly important in the lexical analyzer's initial processing of the input string, but it can be quite useful if a diagnostic message must be issued.

Consider the following attempt at entering a CBASIC expression in response to a prompt from the program generator:

```
X    102Q
```

A stand-alone lexical analyzer would identify X as a variable name and then, on detecting the sequence of digits, would be expecting to find a numeric constant. On detecting the letter "Q", it would note an error. As long as it is possible to defer issuing a complaint about the invalid constant until after syntax analysis has been performed, and as long as priority is given to announcing the missing operator, a sensible error message will result. But making these arrangements may be a complex task. By contrast, if the lexical analyzer is called by the syntax analyzer and reports that it has found a defective constant, the syntax analyzer is in a position to disregard the lexical analyzer's view of the problem. The grammar which drives syntax analysis clearly does not permit even a valid constant here.

The real disadvantage in adopting the stand-alone lexical analysis scheme is not that it cannot match the performance of a system where lexical analysis is subordinate to syntax analysis. Rather, the problem is that the verifier system will be more complex and more awkward because it must work around the natural deficiencies of this organization. The implications of the decision to employ the stand-alone strategy for the lexical and syntax analysis routines will be considered in more detail later in this chapter.

Organizational Constraints Imposed by Other Program Generator Modules

The organization of the verifier system is shaped to a degree by the specialized requirements of some of its callers. In particular, certain program generator modules prompt the user for input which must fall into some subset of valid CBASIC expressions, namely, constants, or identifiers, or expressions which may be used as targets of input or assignments operations. If the expression verifier could identify the members of these three subclasses, the processing performed locally by modules with these specialized requirements could be simplified considerably.

The expression verifier is unquestionably capable of distinguishing among various subclasses of valid expressions. The verifier must analyze the syntactic structure of a candidate expression, and the grammar which guides this analysis could be written so that the subsets of interest to other modules would be identified by their distinctive syntactic features. The cost for this would be a more complex grammar for expressions and hence a larger syntax analyzer. Since storage space is at a premium, it is worth examining alternatives to this approach.

It is possible to determine whether some user-supplied string consists of a single valid identifier or a single valid constant without even invoking the syntax analyzer. The lexical analyzer's function is to recognize exactly these kinds of entities. If any module which must check whether the user's input is an identifier or a constant is permitted to called the lexical analyzer directly, such a module can verify the correctness of the input by examining the token string returned by the analyzer. If the string consists of a single token of the appro-

priate type, then the input is acceptable. This method requires only that the lexical analyzer be accessible directly to routines outside the expression verifier. It requires some additional code in the calling modules to examine the token string; if there are many such modules in the system, it would be space-efficient to have all of them call a single module which in turned called the lexical analyzer and interpreted the result.

The problem of determining whether an expression is valid as the target of an assignment or read operation does not admit of so straightforward a solution. A target is either an isolated scalar variable or a reference to an element of an array. It is easy enough to detect the former, but the latter may include within its subscript list arbitrary numeric expressions. Thus the full expression verifier must be called to check for a valid target. There are three methods which might be used to perform this test:

1. A special target checking module could call the lexical analyzer. If the first token represented an array name, the target checker could then proceed to verify--through local processing and calls to the expression verifier--the correctness of the subscript list.

2. The expression grammar which guides syntax analysis could be written so that targets constitute a recognizable subcategory of the valid expressions. The syntax analyzer could signal whether or not it finds a target.

3. Certain semantic actions might be associated with the syntax analyzer's processing of the candidate expression. These would enable the syntax analyzer to detect a valid target without requiring it to have a special grammar.

The first alternative is too cumbersome and likely to require a substantial amount of additional code. The choice between the second and third options seems clear: the third is likely to make smaller demands for storage. The feasibility of this approach, however, must be

examined more carefully when the design of the syntax analyzer is taken up in a detailed way.

When a program generator module requires that a string supplied by the user fall into a restricted subclass of valid expressions, the handling of errors in the input must be modified somewhat. If the input has a defect but clearly could not fit into the appropriate subclass even if the defect were corrected, the usual error message should not be printed. Rather, a message indicating that the user has not supplied a valid representative of the appropriate subclass is in order. If, for example, the user were asked to supply a target to receive the result of a calculation and entered

A + 35000,

the expression verifier's normal response would be to indicate that 35000 is too large to be a valid integer constant. Under the circumstances, however, such a message is misleading. Instead the system should issue a message indicating that only scalar variables and references to elements of arrays may receive the result of a calculation. The verifier must therefore suppress its normal error response when the caller can accept only a subset of the valid CBASIC expression and the candidate expression clearly does not belong to this subset. This means that the caller must indicate to the verifier, by means of a flag, what restrictions it is imposing upon the valid expressions it will accept. A flag indicating that only a target is acceptable is clearly needed. Moreover, since in some cases a calling module will accept an arbitrary expression of one type only, flags indicating that the candidate expression must be numeric or must be string-valued should also be provided.

The Structure of the Expression Verifier: A Summary

The discussion thus far has established the basic structure of the expression verifier. The design decisions have been guided primarily by the requirement that the system make relatively small demands for primary storage space and, to a lesser degree, by the need to provide certain specialized information to some potential calling modules.

The driver routine, which guides the verifier's processing by calling the major components of the system into primary storage as they are needed, resides permanently in primary storage. This is the routine called by program generator modules which require the expression verifier's services. Thus, in addition to controlling the work of the verifier, it serves as the interface with the rest of the program generator system.

The remaining components of the verifier--those which perform the bulk of the processing--are overlays which are assigned to a single region of primary storage space. They include the lexical analyzer, the syntax analyzer, the error message routine, and the variable declaration module, which is the only error repair facility in the system. Figure 1 depicts the relationship between the verifier's components in a schematic way.

The Syntax Analyzer

Overview of Design Issues

The syntax analyzer is the heart of the expression verifier. It has the task of determining whether a candidate expression--reduced by the lexical analyzer to a stream of tokens--conforms to the rules for

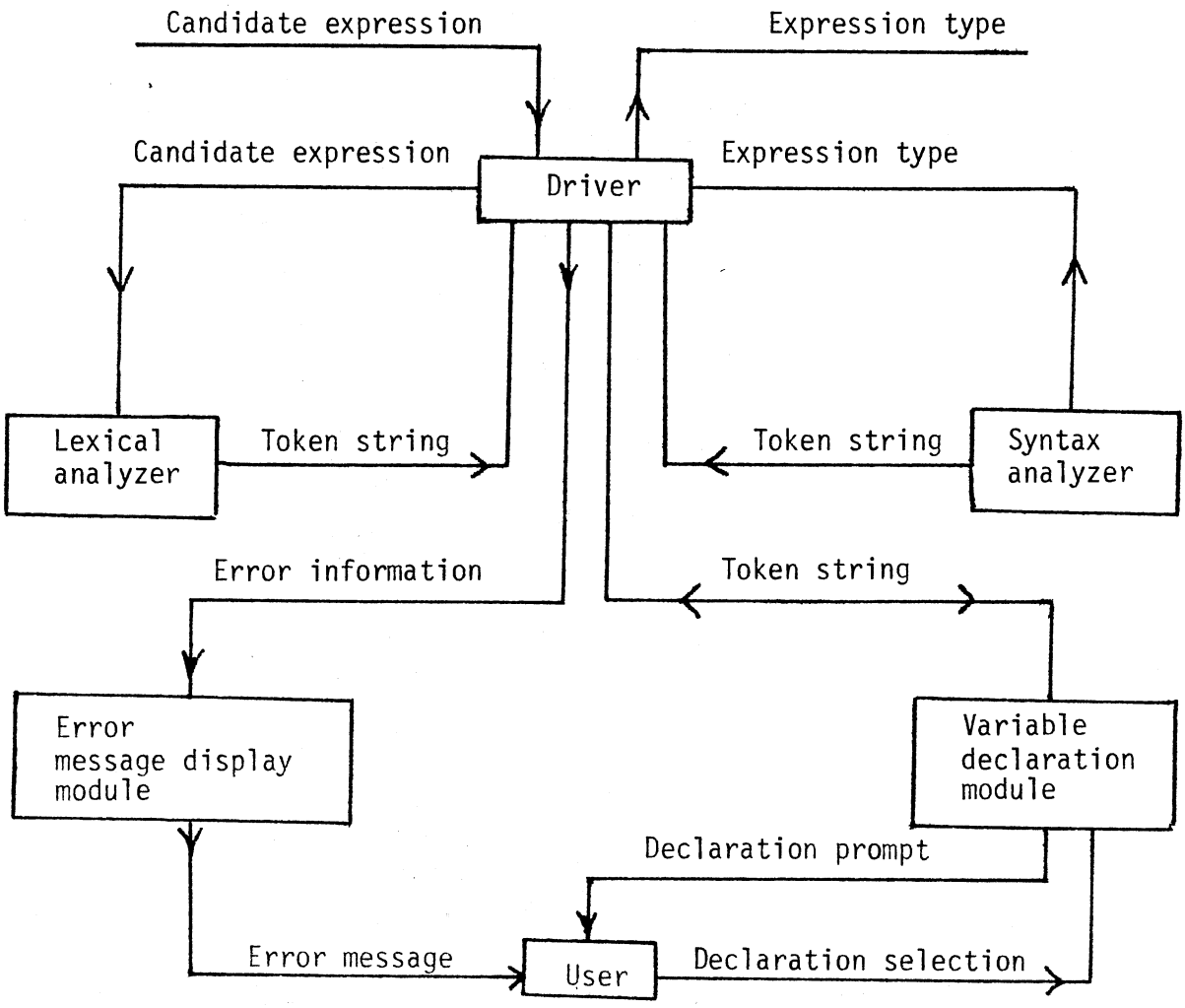


Figure 1. Structure of the Verifier System

forming a CBASIC expression. It must also ascertain whether a valid expression is one which may be used legitimately as the target of an assignment or read operation. In the event that it detects an error, the syntax analyzer must set up a message to be issued by the message display routine.

The syntax analyzer for an expression verifier, like that for a typical compiler for a programming language, is based upon some parsing algorithm. A parser--in the broadest sense of the term--is a recognizer for some language which can be described formally by a grammar. Practical parsing methods are not completely general; each such technique works with some restricted class of grammars. Thus the major design tasks in developing any syntax analyzer are the formal specification of the language which the analyzer is to recognize and the selection of a suitable parsing technique.

The language consisting of well-formed CBASIC expressions may be described by a context-free grammar. An important question arises when one attempts to construct such a grammar, namely, which of the rules for forming expressions are to be encoded into it? Some of these rules are clearly syntactic; the rule which prohibits the sequence of operators "*/" is one example. Such rules are naturally incorporated into the expression grammar. Other rules have a semantic component: for example, an identifier declared to be a one-dimensional array must be followed by a subscript list containing exactly one subscript. This rule and others like it may be encoded into the expression grammar, but they may also be embedded into semantic routines called by the syntax analyzer. The decision between these alternaive approaches is a decision about how one will distinguish between the syntax of the expression language and its

semantics. This choice poses one of the fundamental issues in the design of the syntax analyzer.

Another significant design decision is the selection of the parsing method to be used as the basis for the syntax analyzer. Three criteria are important in choosing from among the variety of techniques available. First, the method must be fast, since the expression verifier will be used in an interactive setting. Second, the method must be modest in its space requirements, since the expression verifier as a whole must be compact. Third, the method must support the production of precise and meaningful diagnostic messages when errors are encountered. Unfortunately, no one parsing technique is clearly superior on all three counts. The design process inevitably involves compromises.

The remainder of this section explores these design issues in more detail and culminates in a specific design proposal. The approach taken here is neither purely empirical nor rigorously theoretical. Rather, the various possibilities for organizing the syntax analyzer are evaluated abstractly insofar as this is fruitful; at critical points, however, the alternatives are compared at a more concrete level.

Syntax and Semantics

The problem of distinguishing between syntax and semantics in an expression language may be illustrated with a language considerably less complex than that for CBASIC expressions. The model language includes string expressions and numeric expressions. The basic elements of the language are identifiers, which may refer to data objects either of type string or of type numeric, the arithmetic operators + and *, and the string concatenation operator +. The type of an identifier must be

declared in advance. Mixed mode expressions are forbidden. The following grammar describes the syntax of this language:

```
<expression> ::= <string expression>
               | <numeric expression>
<numeric expression> ::= <numeric expression> + <term>
                       | <term>
<term> ::= <term> * <factor>
         | <factor>
<factor> ::= idnum
           | (<numeric expression>)
<string expression> ::= <string expression> + <string term>
                      | <string term>
<string term> ::= idstring
                | (<string expression>)
```

(3.1)

Each nonterminal symbol in the grammar is enclosed in brackets (<, >). The terminal symbols idnum and idstring represent identifiers declared to be of type numeric and identifiers declared to be of type string, respectively. In an expression verifier for this language, the lexical analyzer would return one or the other of these tokens upon encountering an identifier.

A parser based upon grammer 3.1 would reject a mixed mode expression such as idstring + idnum because there is no derivation of this string from the start symbol <expression>. The rules related to the types of identifiers are embedded in the formal specification of the syntax of the expression language.

The alternative is to treat identifiers of both types as syntactically indistinguishable. The following grammar does so, but in all other respects it is equivalent to grammar 3.1.

```
<expression> ::= <expression> + <term>
               | term
<term> ::= <term> *<factor>
         | <factor>
<factor> ::= id
           | (<expression>)
```

(3.2)

Here the terminal symbol id represents an identifier of either type.

Grammar 3.2 alone is not sufficient to guide syntax analysis in an expression verifier for the model language. It is possible, however, to transform the grammar into a generalized syntax-directed translation scheme which incorporates the necessary type checking. To do so, each of the nonterminals in the grammar is associated with a field (called a translation) storing the type (numeric or string) of the substring which the nonterminal derives. For each production rule of the grammar, a small program segment is supplied which performs the bookkeeping necessary to keep the type fields current and, where appropriate, checks for mixing of modes or misuse of arithmetic operators. The actions secified by this program segment are performed whenever the corresponding production is used in the parse of a candidate expression [3]. Figure 2 presents a syntax-directed translation scheme for the model expression language, with the program segments--also called semantic actions--expressed in a high-level algorithm language.[3]

Clearly one gains a more compact grammar--one with fewer symbols and fewer productions--at the cost of adding code (in the form of routines which implement the semantic actions) to the syntax analyzer. There is little incentive to use a syntax-directed translation scheme, however, unless the compactness of the grammar is of some tangible benefit. A more compact grammar--all else being equal--reduces the space requirements of the syntax analyzer, since the storage occupied by the production rules is reduced and (often more significantly) since the size of the parsing tables required by any efficient parsing method grows with the grammar which guides the parse.[4] The question of importance to the designer is whether these savings in storage are sufficient

```
< expression> ø ::= <expression>1 + <term>

                { if <expression>1.TYPE = <term>.TYPE then
                     <expression>ø.TYPE := <expression>1.TYPE;
                  else error; }

 <expression > ::= <term>

                { <expression>.TYPE := <term>.TYPE; }

 <term>ø ::= <term>1 * <factor>

                { if <term>1.TYPE and <factor>.TYPE are numeric then
                       <term>ø.TYPE := "numeric";
                  elseif <term>1.TYPE and <factor>.TYPE are string then
                    <term>ø.TYPE := "string";
                  else error; }

 <factor> ::= id

                {<factor>.TYPE := type found by lexical analyzer for id;}

 <factor> ::= (<expression>)

                 <factor>.TYPE := <expression>.TYPE;
```

Figure 2. Syntax-Directed Translation Scheme for the Model Language

to offset the storage occupied by the semantic action routines. Unfortunately there is no generalizable answer.

In the case of the model expression language presented here, it is unlikely that a syntax analyzer based upon the SDTS of Figure 2 would be more compact than one based upon grammar 3.1 The model language, however, does not constitute a realistic basis upon which to decide between the two approaches to performing type checking. Grammar 3.1 does not reflect accurately the extent to which a grammar must be expanded if it is to embody all of the rules related to the types of data objects which may appear in expressions in a practical programming language. Particularly deceptive in the small number of terminal symbols which represent identifiers. Grammar 3.1 has just two. In a more realistic expression language, one might need as many as a dozen different symbols for identifiers and at least that many additional production rules in the grammar. The storage required for the production rules and the parsing tables might well become unacceptably large.

The discussion thus far has overlooked an important aspect of the problem of distinguishing between syntax and semantics in the expression verifier. An expression supplied to the program generator by the user may include one or more identifiers which have not been declared. Given the requirements specified in Chapter II for responding to this kind of problem, the syntax analyzer has two tasks to perform:

1. It must determine whether the expression, apart from its undeclared identifiers, is valid.
2. It must determine the attributes which legitimately may be assigned to each undeclared identifier for which the context implies such attributes.

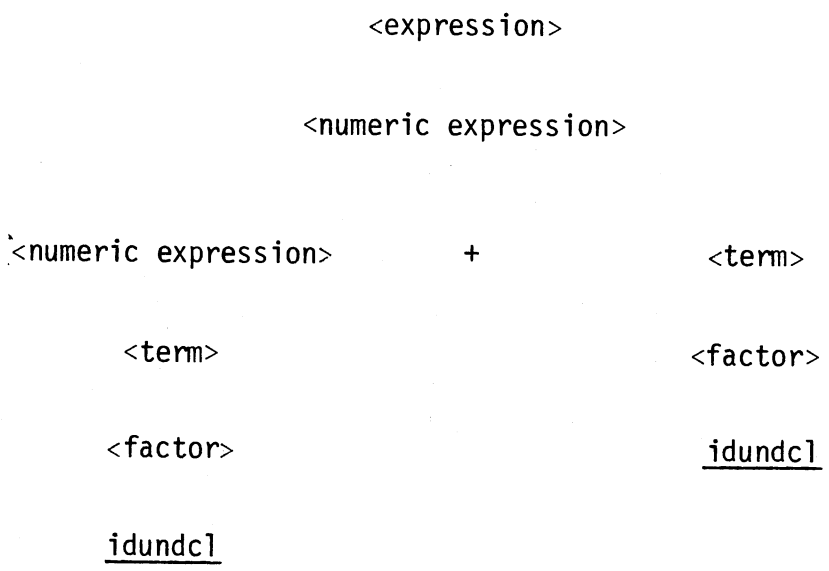
The design issue posed by these requirements is the choice between treating these tasks as semantic actions and treating them, insofar as possible, purely syntactically.

The issue may be explored by considering it in the context of the model expression language. To treat the problem of undeclared identifiers syntactically, one must introduce a new terminal symbol, idundcl, to represent such identifiers. This symbol may appear anywhere in an expression where either idnum or idstring is allowed. The straightforward modification to grammar 3.1 to take account of the new terminal symbol is to add two productions, namely,

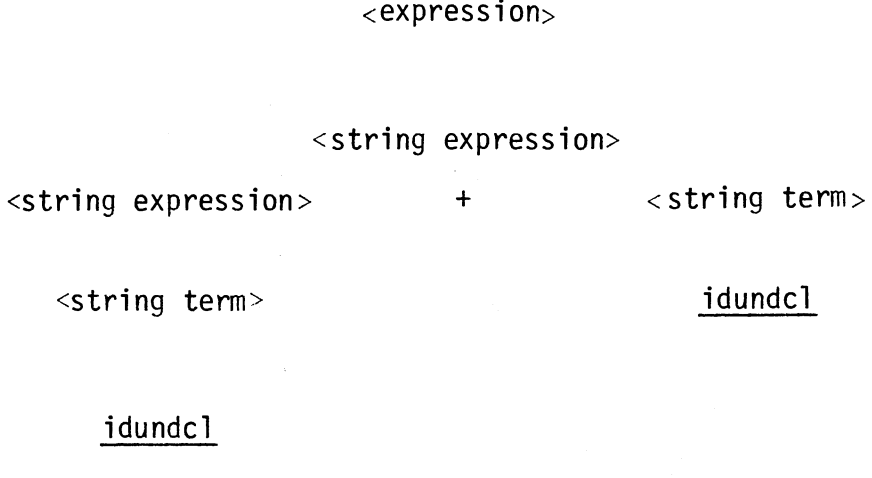
```
< factor> ::= idundcl
        and
<string term> ::= idundcl.
```

This change renders the grammar ambiguous. Figure 3 shows two distinct parse trees for the expression idundcl + idundcl, demonstrating the ambiguity in the grammar. In fact there are two parse trees for every expression of indeterminate type (that is, for every expression which cannot be classified unequivocally as either numeric or string based upon the operators or the types of the identifiers present in the expression).

Since practical, efficient parsing techniques almost invariably demand an unambiguous grammar for the language to be parsed, this straightforward alteration of grammar 3.1 is not an adequate solution to the problem of handling undeclared identifiers. To eliminate the ambiguity, one must treat expressions of indeterminate type as a special case. Moreover, one must add numerous productions to the grammar in order to capture the rule that idundcl may appear anywhere either idnum or idstring is permitted. Grammar 3.3, listed in Figure 4, is an unambiguous grammar for the model expression language which takes account of undeclared identifiers. The number of productions has grown to nearly double that for grammar 3.1.



(a)



(b)

Figure 3. Two Parse Trees Illustrating the Ambiguity in a Simple Extension of Grammar 3.1

By contrast, grammar 3.2, with its six productions, is adequate if the processing needed to permit undeclared identifiers in expressions is incorporated into the semantic actions. The major adjustment required to the syntax-directed translation scheme of Figure 2 is a change in the conditionals which compare the types of already-parsed subexpressions. Strict equality of types is no longer required, for example, for the <expression> and the <term> of the first production rule. If the types are unequal but one is of indeterminate type, the subexpression <expression> + <term> is still valid. The effect of the modifications to the semantic actions is to make the code for them somewhat larger.

Thus the first task associated with processing an expression containing undeclared identifiers-accepting the expression if its only defects involve these identifiers--can be accomplished either syntactically (at the cost of a significantly larger grammar) or semantically (with somewhat larger semantic routines). The second task--establishing the attributes of the identifier if they are clear from its context in the expression--cannot be performed without some recourse to semantic routines. At the very least it will be necessary to record the attributes as they are determined, and this will require that some recording routine be invoked.

Consider the simple expression A + B, where A has been declared to be a string variable but B is undeclared. The expression is valid only if B is a string value. The expression verifier must take note of this fact so that the repair module which allows the user to supply a declaration for B restricts the user's options to declaring B to be a scalar variable of type string or abandoning the expression. In an expression verifier for the model expression language of grammars 3.2 and 3.3, processing would take one of the following forms:

```
<expression> ::= <string expression>
               | <numeric expression>
               | <indeterminate expression>

<numeric expression> ::= <numeric expression> + <term>
                       | <numeric expression> + <indeterminate term>
                       | <indeterminate expression> + <term>
                       | <term>

<term> ::= <term> * <factor>
         | <term> * <indeterminate term>
         | <indeterminate term> * <factor>
         | <factor>

<factor> ::= idnum
           | (<numeric expression>)

<string expression> ::= <string expression> + <string term>
                      | <indeterminate expression> + <string term>
                      | <string expression> + <indeterminate term>
                      | <string term>

<string term> ::= idstring
                | (<string expression>)

<indeterminate expression> ::=
                    <indeterminate expression> + <indeterminate term>
                  | <indeterminate term>

<indeterminate term> ::= idundcl
                       | (<indeterminate expression>)
```

Figure 4. Simple Expression Grammar, Modified to Include Undeclared Identifiers

1. If grammar 3.2 is the basis for the verifier, the lexical analyzer would reduce the input text to the token string id + id. The parser would eventually use the production <expression>$^{\emptyset}$ ::= <expression>^{1} + <term>. Since the first id is of numeric type while the second is undeclared, <expression>^{1}.TYPE is string while <term>.TYPE is indeterminate. The semantic action to be taken here consists of setting <expression>$^{\emptyset}$.TYPE to string and noting that <term>.TYPE--and hence the type of the identifier B--must be string as well.

2. If grammar 3.3 is the basis for the verifier, the lexical analyzer would reduce the input text to the token string idstring + idundcl. The parser would eventually use the production <string expression> ::= <string expression> + <indeterminate term>. No semantic action is required to test for a mixed mode operation, since the grammar assures that these cannot be parsed. But some action is required to take account of the fact that the indeterminate term --and hence also the identifier B which constitutes it--must eventually be made to be of type string.

Thus even when the rules for types are encoded as much as possible into the grammar for the expression language, it is impossible to avoid semantic processing altogether.

The foregoing discussion has shown that there are indeed two workable approaches to handling type information in an expression verifier. The choice between the two is among the fundamental issues facing the designer of a verifier. Unfortunately, neither method is clearly superior. For the present project, the decision is to treat all type information semantically; that is, the verifier's syntax analyzer will be based on a syntax-directed syntax scheme much like that of Figure 2.

This approach has the benefit, previously noted, of keeping the grammar upon which the syntax analyzer is based relatively small. This, in turn, reduces the size of the parsing tables required by any of the common table-driven parsing methods. This savings in storage at least partially offsets the additional storage required for the semantic routines. A second benefit of this scheme is its flexibility. If one

wished to alter the verifier so that it could distinguish between expressions of type integer and those of type real, one could so at the modest code of expanding the semantic routines somewhat. By contrast, if type checking is performed syntactically, such a modification would require a substantial expansion of the grammar and consequently of the parsing tables. Finally, adopting the semantic approach to type checking has the further advantage of giving the designer a wider range of options in the choice of parsing techniques. In particular, it leaves open the possibility of basing the syntax analyzer on an LL(k) grammar. It is notoriously difficult--if not impossible--to encode type rules for any realistic expression language into an LL(k) grammar.[5] Yet, as a later discussion shall show, LL(k) parsers are among the more attractive choices in the present application.

Parsing Methods: A Survey

Numerous techniques have been devised for parsing context-free languages. Although virtually any of them could be used to recognize syntactically well-formed CBASIC expressions (provided that the expression grammar is cast in a suitable form), not all of them can meet the requirements for speed, compactness, and ability to isolate errors, which are of great importance in the present application. While an exhaustive survey of parsing techniques is not appropriate in a design study which aims at specifying a practical product within a reasonable amount of time, a review of some of the methods which hold out some prospect of being suitable is fruitful. The present section provides such a review.

The parsing methods which involve backtracking (that is, those which may require multiple passes over portions of the input string) are very attractive because of their compactness. A backtrack parser requires no parsing table; it needs only a representation of the production rules for the grammar. It operates by attempting to construct a derivation of the input string through an application of the production rules on a trial-and-error basis: a potential derivation is pursued until it becomes inconsistent with the string being parsed, at which point the parser backs up in the input string and attempts an alternative derivation. A backtrack parser may work either from the bottom up, in which case it attempts to construct a derivation of the input string in reverse (working from the input string back to the start symbol for the grammar), or from the top down, in which case it seeks a series of production rules which leads from the start symbol to the input string [2].

Backtrack parsers are relatively easy to implement because they impose few restrictions on the grammars for the languages they are to parse. They are economical in their use of storage space, requiring in addition to the code which implements the parsing algorithm only a stack (whose maximum depth is, in the worst case, a linear function of the length of the input string) and a representation of the production rules for the grammar. In a microcomputer environment, this compactness is a great virtue.

Unfortunately, backtrack parsing has some serious drawbacks. First, in the worst case its execution time is an exponential function of the length of the input string [2]. Surely it is a fundamental maxim of interactive system design that one ought not to invoke an algorithm of exponential time complexity while the user is waiting at the terminal!

Thus the use of a backtrack parser in an expression verifier for an interactive program generator is open to serious criticism. Second, a backtrack parser provides virtually no means of issuing meaningful diagnostic information; it is incapable of precisely identifying the location and nature of a syntax error [2]. This technique is therefore not well suited for use in the expression verifier under consideration here.

Backtrack parsing has its place in expression verification in program generator systems, despite the fact that it does not meet the requirements imposed for the present design effort. Where space is severely restricted, backtrack parsing may be the only workable alternative. When detailed error messages are not required--as, for example, when the users of the program generator are experienced programmers--backtrack parsing may be an acceptable choice. The time complexity of the method remains something of a problem in either case.

Of the parsing methods which do not involve backtracking, several appear to be worthy candidates for use in the expression verifier's syntax analyzer. Two are top-down methods: recursive descent parsing and its table-driven variant, predictive parsing. The others are bottom-up techniques: operator precedence parsing and LALR parsing.

A recursive descent parser is a collection of mutually recursive procedures, each of which corresponds to a nonterminal symbol of the grammar for the language to be parsed and recognizes strings which that nonterminal derives. The procedures represent a kind of encoding of the production rules for the grammar. A recursive descent parser attempts to construct a derivation of the input string from the start symbol in which, at each step, the leftmost nonterminal is expanded, that is,

replaced by the left-hand side of one of the production for which it is the right-hand side. In order to avoid backtracking, the parser must be able to choose (from among the several productions for which the non-terminal being expanded is the right-hand side) that production which will put the parser on the path towards deriving the input string if in fact it belongs to the language. The selection is made based upon an examination of the next one or more characters of the unexpended input string.

Recursive descent parsers work with a subset of the context-free grammars known as LL(k) grammars. These grammars are unambiguous and free of left-recursion.[6] They possess certain other properties which guarantee that at any point in its reconstruction of the leftmost derivation for some string in the language it recognizes, a recursive descent parser will be able to select the one applicable production by examining the next k characters of the input string. One potential difficulty in using a recursive descent parser is that it is not always easy--or even possible--to construct a suitable grammar for the language to be parsed [4].

The major obstacle to employing a recursive descent parser in the present application, however, is the technique's reliance on recursive procedures. Although programming languages which support recursion have been implemented for microcomputers, the cost of using recursive procedures--in terms of the storage required for the activation records for pending invocations of procedures--is dangerously high in an environment in which storage is at a premium.

Another top-down parsing technique, which Aho and Ullman [2], call k-predictive parsing (or simply predictive parsing),[7] is similar in

approach to recursive descent but avoids the use of recursive procedures. Instead, a predictive parser maintains a stack which records the history of its parsing decisions. A table--indexed on one dimension by the nonterminal symbols of the grammar and on the other by strings of terminals--guides the selection of the production to be used at each step of the parse. This parsing table may be constructed using an algorithm which examines the production rules for the grammar to be parsed. As with the recursive descent technique, the grammar must be LL(k), for some k greater than or equal to one. For many applications, an LL(1) grammar is adequate. A parser for a language defined by such a grammar need examine only the first character of the unexpended input string to determine which production to use in expanding the leftmost nonterminal in the sentential form it has thus far derived. The parsing table which guides this determination is indexed by the nonterminals of the grammar on one dimension and by the terminal symbols on the other. Thus for a grammar with a modest vocabulary, the table is quite compact [2].

In addition to its relatively modest storage requirements, predictive parsing offers two benefits equally important in the present application. First, a predictive parser is fast--it runs in time proportional to the length of the input string [2]. Second, such a parser is able to isolate an error in the input in such a way that a reasonably precise diagnostic message may be issued [3]. In short, a predictive parser is well-suited to serve as the basis for the expression verifier's syntax analyzer.

Appendix B lists the production rules of an LL(1) grammar which describes the syntax of CBASIC expressions. One feature of this grammar

has important implications for the design of a syntax analyzer based on a predictive parser: the production rules impose a rather unnatural--though perfectly correct--structure on expressions. In particular, the grammar does not support the ordinary algebraic view of an expression as a collection of subexpressions connected by operators. For example, grammar 3.2--which is not LL(1)--has this relatively natural description of an arithmetic term:

<term> ::= <term> * <factor>

In the LL(1) grammar of Appendix B, the corresponding description is more awkward:

<term> ::= <factor> <term more>

<term more> ::= mulop <factor> <term more>

|null

If this were merely an aesthetic matter, it would not be worth mentioning. In fact, the manner in which the LL(1) grammar breaks up subexpressions makes semantic analysis--in the present application, verification that the types of operands are correct--somewhat complicated. To verify that a multiplication does not involve mixed or illegal types, for example, the semantic routine must have access to information about the types of both of the operands. In a predictive parser, this information does not appear together on the parser stack. Typically, if semantic processing of complete subexpressions is required, the parser must maintain auxiliary stacks for operators and operands. Such a scheme would be necessary in the syntax analyzer for an expression verifier if it were based on a predictive parser. It is worth nothing that in its need to group together the operands and operator for each subexpression in an expression, the semantic (type verification) processing of the

expression verifier resembles the intermediate code generation phase of a compiler for a language which supports arithmetic and string expressions. Pyster [20] describes a code generation scheme for a compiler which parses according to an LL(1) grammar. This scheme might serve as a model for the semantic action routines for the expression verifier's syntax analyzer.

Certain non-backtracking bottom-up parsers for languages consisting of expressions may be based on grammars resembling grammar 3.2. Such a parser supports straightforward type verification, since it can call for semantic processing when a complete subexpression is at the top of its stack.

Bottom-up parsers attempt to construct a derivation for the input string in reverse. The basic operations in such a parser are shifting a symbol from the input string onto the parser stack and reducing a sequence of symbols on the stack by substituting a nonterminal for the sequence, where this nonterminal is the lefthand side of a production for which the sequence of symbols is the right-hand side. When a series of parser moves leaves the grammar's start symbol alone on the stack with no more characters to be examined in the input string, the parser has accepted the string. The fundamental problem in constructing a bottom-up parser is devising a method for guiding the parse: given an arbitrary configuration of the parser, it is necessary to specify whether the parser is to shift or to reduce (and, if it is to reduce, the parser must identify the production to be used). Non-backtracking bottom-up parsers typically make use of a parsing table to direct the parse. These parsing techniques employ various methods for constructing the tables and apply to various subsets of the context-free grammars [3].

In operation-precedence parsing, the shift-reduce decision (as well as the selection of the production by which to make a reduction when this is required) is guided by three disjoint relations--called precedence relations--on the set of terminal symbols for the language to be parsed. In the parse of an input string, the relation which obtains between the top terminal symbol on the parsing stack and the first symbol in the unprocessed portion of the input string determines whether the parser is to shift the current input symbol onto its stack or to reduce some sequence of symbols already at the top of its stack to a single nonterminal. When a reduction is required, the precedence relations which obtain between pairs of terminal symbols on the stack determine the extent of the handle (that is, the sequence of grammar symbols to be reduced). Given a grammar in which no two production rules have identical right-hand sides, determining the handle amounts to selecting the production by which the reduction is to be made [3].

Although operator precedence parsing applies to a limited subset of context-free languages, languages which define sets of arithmetic expressions typically fall into this subset. On two of the three criteria of importance in choosing a parsing method the expression verifier, speed and compactness, the operator precedence technique excels.

Operator precedence parsers are typically fast, since there is no backtracking involved. Execution time may be adversely affected by the requirement that the handle be matched to the right-hand side of some production when a reduction is called for, but unless the grammar is quite large or the matching algorithm especially inefficient the parser should perform satisfactorily.

Parsers employing the operator precedence technique have modest storage requirements. Like the other types of parsers under consideration here, operator precedence parsers require a parsing stack and a representation of the production rules for the grammar which generates the language to be parsed. They require, in addition, some representation of the precedence relations. A very compact scheme for storing these encodes them into a pair of precedence functions; two vectors with n integer entries each are sufficient to store the precedence information for a language wih n terminal symbols. Unfortunately, this method delays the detection of errors and complicates the problem of producing adequate diagnostic messages. An alternative representation for the precedence relations employs a two-dimensional array indexed along each dimension by the terminal symbols of the grammar. Each entry in the array indicates which, if any, of the three precedence relations obtains between the pair of terminal symbols which index the entry [3]. This method, while more demanding of storage space, takes advantage of the full power of the parser to detect errors and is the technique of choice in an application where error handling is of central concern.

Since predecence relations are defined only on the set of terminal symbols, nonterminal symbols have no influence on the parse of an input string. If the grammar for the language to be parsed contains single production (that is, productions of the form A ::= B, where A and B are both nonterminals), an operator precedence parser may fail to accept a string in the language because it fails to make an essential reduction by a single production. This problem can be circumvented by adding to the parser special code--peculiar to the grammar for the language being parsed--to test, at each reduction made by the parser, whether a further

reduction by a single production is appropriate. Alternatively, one can treat the nonterminals of the grammar as indistinguishable and use a single symbol to represent any nonterminal on the parser stack. Then single productions, since they are used to reduce one nonterminal to another, are irrelevant. Using this scheme, one can be certain that the parser will accept all strings in the language generated by the grammar, but one cannot be sure that certain strings not in the language will not also be accepted [2]. Since grammars for the arithmetic and string expressions suitable for use with an operator precedence parser rely on single productions to encode the rules for the associativity of binary operators and for the order of evaluation of subexpressions, any attempt to use operator precedence parsing in the expression verifier must take account of this problem.

An operator precedence parser detects syntax errors in a candidate sentence in two distinct ways. First, it may find that no precedence relation obtains between the topmost terminal symbol on its stack and the current input symbol. In this case, the table storing the representation of the precedence relations might contain an entry which indicates what action is to be taken to recover from the error. Second, the parser could find, on making a reduction, that the handle it has isolated matches none of the right-hand sides of the production rules of the grammar. Here the central problem in error recovery is determining which right-hand side the handle most nearly resembles [3].

Fortunately, the defective handles which may be isolated by an operator precedence parser for a language consisting of arithmetic and string expressions fall into a small number of categories. Typically

such defective handles reflect missing operands in the input string. This is not surprising since in a typical expression grammar (e.g., grammar 3.2) the operands in a subexpression are represented by nonterminals (as in the production <term> ::= <term> * <factor>), and nonterminals have no bearing on the parser's determination of the extent of the handle. It is relatively easy to isolate these kinds of errors and to take the appropriate actions to recover from them [3]. One class of defective handles which a parser for CBASIC expressions might isolate is infinite. These handles are found when a subscript or argument list contains too many subscripts or arguments or when a list contains consecutive commas. This condition is not difficult to detect, but producing a meaningful diagnostic message in response to it adds complexity to the syntax analyzer.[8]

In sum, the operator precedence technique is a reasonable choice of parsing method for the expression verifier's syntax analyzer. It offers a fast and compact parser. It has the disadvantage of requiring some elaborate ad hoc extensions for handling single productions and for processing certain kinds of errors.

LR parsing offers the most straightforward method for syntax analysis in the expression verifier. Like the operator precedence technique, LR parsing works with a grammar which reflects the natural structure of expressions. But it does so with none of the awkwardness characteristic of operator precedence parser--single productions pose no problem for an LR parser, and error handling is completely consistent. The benefits of LR parsing come at the price of somewhat larger parsing tables. Aho and Ullman [2] describe the theory behind the technique; Backhouse [4] offers a different and occasionally more lucid treatment of the topic.

Aho and Ullman [3] elsewhere provide a practical guide to constructing LR parsers.

The lookahead--LR (LALR) technique offers considerable power (that is, it can be applied to a large class of context-free languages) while requiring relatively compact parsing tables. The most general LR method, so-called canonical LR parsing, often requires tables so large--even for languages with a few dozen production rules--as to be impractical [3]. The present study, therefore, focusses on the LALR technique. In particular, LALR(1) parsing, in which a parsing decision (the choice between shifting a symbol from the input string onto the parsing stack and reducing a sequence of symbols already on the stack) is guided by the contents of the stack and the first character of the unexpended input string, is to be explored here.

The grammar of Appendix A is an LALR(1) grammar. It is an unambiguous grammar which encodes all of the rules for the associativity of binary operators and precedence rules (that is, rules for the order of evaluation) for all operators. The numerous single productions serve primarily to establish these rules. Without them, the grammar would be ambiguous, and no ambiguous context-free grammar is an LR grammar. The grammar of Appendix C, by contrast, is ambiguous, precisely because it does not encode the precedence and associativity rules for the binary arithmetic operators. It has fewer production rules and a smaller set of nonterminal symbols than the grammar of Appendix A. The grammar of Appendix C is of interest here because, despite its ambiguity, it can be parsed by an LR parser. The ambiguous character of the grammar is reflected by the fact that the algorithm for constructing LALR(1) parsing tables detects conflicting parsing decisions in certain situations.

Specifically, because certain precedence and associativity rules are not encoded into the grammar, it is sometimes not clear whether the parser should reduce a subexpression on its stack or shift the next input symbol (which, in the situations where the conflicts arise, is always a binary arithmetic operator) onto the stack. The table-building algorithm therefore attempts to insert two entries--one calling for a shift, the other for a reduction--into one position of the table. It is possible to base an LALR(1) parser on the grammar of Appendix C because it is possible, based on the precedence and associativity rules given in the definition of the expression language, to resolve the conflicts by selecting the appropriate parsing table entry where the table-construction algorithm finds two [3]. This technique does not apply to all ambiguous grammars, but it works for the present application. The advantage is that it saves storage space by reducing the number of production rules (which must be stored, in some form, within the syntax analyzer) and by reducing the size of the parsing tables.

Except for the backtracking methods, any of the parsing techniques reviewed here represent reasonable alternatives for implementing a syntax analyzer for the expression verifier. For this project, the LALR(1) method, relying on the ambiguous grammar of Appendix C, will provide the basis for the design. Its principal virtue lies not in superior performance on the three criteria of importance to the expression verifier--speed, compactness, and ability to provide adequate error diagnostics--but in its naturalness. The LALR(1) grammar, unlike the LL(1) grammar, reflects the structure of expressions in a way which conforms to the ordinary algebraic division of expression into subexpressions and which readily supports the kinds of semantic processing required

in the present application. And, unlike the operator precedence technique, LR parsing is not burdened with the need for ad hoc extensions to permit the use of a grammar with single productions and to process certain classes of errors. In brief, by selecting the LALR technique for this application, the designer sidesteps--or permits the implementor to sidestep--the need to work around the special problems posed by the other methods.

The Semantic Actions

The syntax analyzer for the expression verifier is to be guided by the syntax-directed translation scheme listed in Appendix D. The semantic actions associated with the production rules implement three types of processing: first, they include tests which verify that subexpresions are of the correct type given the operations to be performed; second, they provide a mechanism for establishing the attributes of undeclared variables whenever their context makes this possible; and third, they enable the syntax analyzer to determine whether the candidate expression is valid as the target of an assignment or read operation.

Type checking employs an expanded version of the processing described in Figure 2 for the model expression language. The greatest source of additional complexity is the treatment of argument lists for CBASIC built-in functions. Each such function requires a definite number of arguments of the appropriate types in a prescribed order. The type verification logic invoked when a reduction is made using one of the productions

<factor> ::= id (<expression>),

<factor> ::= id (<expression>, <expression>), or

<factor> ::= id (<expression>, <expression>, <expression>)

must determine what sort of subscript or argument list id requires. This information comes from the symbol table entry for the identifier represented by id, information which the type checking routine obtains either by accessing the symbol table or from data provided by the lexical analyzer as part of the token string for the candidate expression. The type checking routine must then ensure that the type of each expression in the list within the parentheses is acceptable. The CBASIC function MID$, for example, which returns a substring of its string argument, takes an argument list consisting of a string expression followed by two numeric expressions. The type verification logic must enforce this requirement.

The syntax analyzer's handling of undeclared variables is based upon the fact that the context in which such a variable appears often makes it obvious what the variable's attributes must be if the candidate expression is to be valid. The variable's position in a subscript or argument list, or the operator being applied to the variable, or the type of the variable's companion operand in a subexpression which joints two operands with a binary operator, might serve as the clue which establishes an undeclared variable's type. The number of dimensions for the variable is easily ascertained from the subscript list, if any, which follows the identifier.

The principal difficulty in establishing the attributes for individual undeclared variable is that the syntax analyzer has the information which makes it possible to determine a type only after it has

reduced the variable to a <factor>, a <term>, or some other nonterminal. To put it somewhat differently, the syntax analyzer deals with subexpressions of indeterminate type and not directly with undeclared identifiers. Consider, as an example, the expressions

A * B

and

A * (B + C)

where A is declared to be a scalar numeric variable and B and C are undeclared. In both cases, the syntax analyzer's first opportunity to establish the type(s) of the undeclared variable(s) comes when it is about to make a reduction using the production

<term> ::= <term> * <factor>.

At this point in the parse of the candidate expression, the translation associated with <factor> which gives its type (and which is denoted by <factor>.TYPE) has the value "indeterminate." But because <factor> represents a subexpression which is an operand of the arithmetic operator *, its type must be numeric if the expression is to be valid. The problem lies in taking this newly-acquired information about the subexpression and applying it to the variables which it includes. From an intuitive point of view, it is clear that in the case of the first expression, <factor> is associated with the single variable B, while for the second expression <factor> stands for the subexpression (B + C). Thus for the first expression the syntax analyzer must record that B must be a scalar numeric variable, and for the second it must note that both B and C must be scalar numeric variables. The syntax analyzer must be provided with a mechanism which can perform this sort of processing in a systematic way.

The syntax-directed translation scheme of Appendix D supports such a mechanism. It maintains a linkage between any nonterminal which represents a reduced subexpression and the sequence of tokens which constitute that subexpression using the translation fields FIRST and LAST. These are pointers to the first and last tokens, respectively, in the sequence making up the subexpression. FIRST and LAST are kept on the parsing stack, along with the nonterminal symbol itself and the translation field TYPE. For any nonterminal on the stack whose TYPE has the value "indeterminate," the sequence of tokens marked out by its FIRST and LAST fields contains at least one token representing an undeclared variable. Whenever the context makes it clear what the type of a subexpression represented by such a nonterminal must be, this newly-acquired information applies to those identifiers represented by the tokens between FIRST and LAST.

When a subexpression whose type has just been established from its context consists of only a single token (namely, an undeclared variable), the syntax analyzer need only mark the token by changing its type information field from "undeclared" to "must be string" or "must be numeric." When the subexpression consists of several tokens of which more than one represent undeclared variables, however, it is not clear which tokens should receive the new type information. Fortunately, a very simple rule applies: Whenever the type of a subexpression is established by context, all of the subexpression's undeclared variables for which types have not been established already receive the new type information. This scheme is effective because the parser works in a bottom-up fashion, analyzes the candidate expression according to its natural division into subexpressions, and takes account of the precedence and associativity of the operators.

In establishing the attributes for undeclared variables, the syntax analyzer must check for inconsistent usage of the same undeclared name within the candidate expression. In the input string

A * A (I),

for example, the name A is used for both a scalar variable and for an array. If A has not been declared, the syntax analyzer will discover conflicting requirements for the attributes of A. In a case such as this, the expression verifier's response should be to signal the inconsistent use of the name rather than to prompt for a declaration.[9] To detect this sort of error, the syntax analyzer must maintain some record of the fact that it has established attributes for an undeclared name. Whenever it attempts to establish the attributes for an undeclared variable, it must check that there is no prior inconsistent usage of the name. Probably the most straightforward method for accomplishing this is by creating a symbol table entry for a name when its attributes are first established. The symbol table is consulted each time the attributes are about to be established for an undeclared name to ensure that there is no inconsistency of usage.[10]

In order to test for inconsistent usage of undeclared names, the syntax analyzer must have access to the names themselves and not merely to the tokens emitted by the lexical analyzer. This linkage to the text entered by the user is accomplished by requiring that the lexical analyzer supply, as part of the information accompanying each token it emits, a pointer to the beginning of the corresponding section of the user's input string and the length of this section. The syntax analyzer can determine the name associated with an identifier token by extracting the appropriate substring of the input.

The third significant task accomplished by the semantic actions of the syntax-directed translation scheme of Appendix D is the determination of the candidate expression's validity as the target of an assignment or read operation. An expression may be so used if it consists of a single scalar variable or a single reference to an element of an array. The flag VALID_AS_TARGET will be set to "true" if the expression is indeed valid as a target. The syntax analyzer relies on the fact that any expression which may be used as a target has a rightmost derivation which begins

<expression> ==> <relation> == <simple expression>

==> <element>.

Thus the parse of any such expression ends with a sequence of reductions by the production rules

<simple expression> ::= <element>

<relation> ::= <simple expression>

<expression> ::= <relation>.

Further, any expression which may be used as a target has a derivation which makes use of at least one of the following production rules:

<element> ::= id

<element> ::= id (<expression>)

<element> ::= id (<expression>, <expression>),

<element> ::= id (<expression>, <expression>, <expression>).

The semantic actions set the flag VALID_AS_TARGET when any of these four production rules is used to make a reduction. The flag's setting is unchanged when any of the three single productions listed above is used. The flag is reset when any other production is used. The result of the actions is that the flag VALID_AS_TARGET is correctly set at the conclusion of syntax analysis.

Summary of Design Decisions

The syntax analyzer for the expression verifier is to be based upon an LALR(1) parser. Although other parsing techniques are reasonable choices in this application, the LALR(1) method represents a natural and straightforward approach to analyzing the syntactic structure of candidate expressions.

Much important work is assigned to the semantic action routines called by the parser. Testing for mixed mode operations and for compatibility of operators and operands is performed by such routines. Semantic actions play a central role in establishing the attributes of variables which have not been declared previously. Comparatively simple semantic actions are used to determine whether a candidate expression is valid as the target of an assignment or read operation.

The implementation of a syntax analyzer according to the design proposed here poses two significant challenges. First, the implementor must work out specific error-handling strategies corresponding to each empty entry in the parser's action table. Each such strategy involves setting up an error message which conforms to the guidelines established in Chapter II and recovering from the error (by altering the configuration of the parser) so that the remainder of the candidate expression may be analyzed. Second, the implementor must code the semantic action routines as compactly as possible. These routines have many responsibilities, and without considerable care in programming they could grow to be unacceptably large.

The Lexical Analyzer

The lexical analyzer's function is to recognize the basic entities of the expression language (identifiers, operators, and other symbols) present in the candidate expression. For each such entity, the lexical analyzer emits a single token. The syntax analyzer then parses the string of tokens corresponding to the candidate expression rather than the candidate expression itself. In the present application, the syntax analyzer performs some semantic processing which requires more information than is provided by the tokens themselves; thus the lexical analyzer's output includes, in addition to the tokens, information associated with each token (for example, the attributes of an identifier).

The information requirements of the syntax analyzer dictate the output of the lexical analyzer. For each basic entity it encounters, the lexical analyzer must, of course, emit a token. The token may be a small integer value which is a code known to the syntax analyzer and which identifies what sort of entity which has been found. Along with this, the lexical analyzer should indicate the position within the user input string at which the entity begins and the length (in characters) of the entity. This information establishes a link between the token and the user's input. Such a link is useful in providing precise error messages. When the token represents an identifier or a constant, the syntax analyzer must have information about its type. When an identifier token represents a CBASIC built-in function name, the type information must indicate not only what type of value the function returns but also must encode an indication of the types of arguments the function requires. The type field supplied by the lexical analyzer may be a small integer value. A range of these values would indicate a numeric-value entities;

another range, string-valued entities, yet another, entities of uncertain type. For built-in functions, the type code could index a table giving descriptions of valid argument lists. Finally, the syntax analyzer, in its semantic processing, makes use of the number of dimensions declared for an array and the number of arguments permitted for a function. This information may be embedded in the type code, but processing will be simplified if it is made available in a separate field.

Thus for each basic entity it encounters, the lexical analyzer emits a package of information. Since the analyzer processes the entire candidate expression, it often will emit more than one such package. For space efficiency, storage for token packages should be allocated as they are needed at execution time. The packages may be linked together into a list, and the lexical analyzer need return to its caller only a pointer to the head of this list. Figure 5 shows three types of nodes which may appear on a list of tokens: one for identifiers, one for constants, and one for other language entities (operators and "punctuation"). Four fields are common to all three types: LINK, which points to the next token package on the list; TOKEN_CODE, which indicates the kind of entity represented by the token; START, which gives the position in the input string of the first character of the text corresponding to the token; and LENGTH, which gives the length of the text corresponding to the token. In addition, the nodes for identifier and constant tokens include the field TYPE, which provides type information for the syntax analyzer. The node for identifiers has, in addition, a field storing the number of elements--if any--which must appear in a subscript or argument list following the identifier.

(a) Identifiers	(b) Constants	(c) Others
LINK	LINK	LINK
TOKEN_CODE	TOKEN_CODE	TOKEN_CODE
START	START	START
LENGTH	LENGTH	LENGTH
TYPE	TYPE	
ELEMENTS		

Figure 5. Nodes for the Token List Emitted by the Lexical Analyzer

The elementary entities which may appear in a CBASIC expression--identifiers, constants, operators, and other symbols--may be described formally by regular grammars. They may therefore be recognized by finite automata. The lexical analyzer for the expression verifier is to be based upon a deterministic finite automaton which recognizes these entities. The lexical analyzer must also include certain semantic routines, associated with the various states of the automaton, which enable it to collect the information to be placed into token packages. The semantic routines may also test that entities in the input string meet certain requirements not conveniently encoded into the automaton itself (for example, the limits on the magnitudes of numeric constants). Aho and Ullman [3] offer practical guidance in the implementation of lexical analyzers based upon finite automata.

The lexical analyzer for the expression verifier requires some sophistication in its handling of errors. On encountering an error in the input string, it must continue processing until it encounters a character which serves as a delimiter for the defective entity, set up an error message (if an error with an equal or higher announcement priority has not been encountered already), emit a token package, and resume processing for the remainder of the input string. To implement this error handling strategy, the automaton guiding lexical analysis is to be provided with states in which the automaton, having found a defect in an entity, scans the input string until it encounters a delimiter. On encountering a delimiter, the automaton enters a state for which the associated semantic action is a routine which sets up an error message and adds a node to the token string.

An important feature of the present design for an expression verifier is that no special tokens are required to represent defective lexical entities. In constructing a token package for a defective entity, the lexical analyzer uses the token code it would have supplied had it found a correct version of the entity. Thus, on encountering the defective variable name "customer_name", the analyzer would emit an identifier token. If the expression verifier found no other error with greater priority for announcement, it would issue the error message set up by the lexical anlyzer when it encountered the defective name.

There are two complications associated with this scheme for error handling. First, there is the problem of assigning a type to a defective identifier. Since it cannot have been declared, its natural type is "undeclared." In order to save some unnecessary processing in the syntax analyzer, however, it is probably best to distinguish defective identifiers from well-formed but undeclared identifiers. This can be done by creating a new type ("defective") which is treated as undeclared but for which the processing involved in establishing attributes is bypassed. Second, there may be invalid sequences of characters in the input string which cannot be said to be "defective versions" of any of the language's basic entities. There are sequences beginning with a character which can begin no valid entity. There are at least two alternatives for handling these:

1. The lexical analyzer could set up an error message but emit no token package. The drawback here is that the syntax analyzer may override this message with a complaint about a missing entity.

2. The lexical analyzer could treat such sequences by assuming that they are defective instances of some class of entities. In fact it could use the first character of the sequence to determine to which class it would be assigned. This means that the implementation must build

in assumptions about what the user intends by a sequence beginning with a certain character.

The second alternative seems best. Although it may occasionally give rise to a less than precise message, it does not effectively ignore the invalid sequence and thus is less likely to produce confusion than the first technique.

Implementing the lexical analyzer would require a substantial effort. In addition to devising the finite automaton and programming it as compactly as possible, the implementor must provide semantic routines. The present discussion has indicated the basic technique to be used by the analyzer and has specified how it is to present the results of its processing to its caller, but much of the detail is left to the implementor.

Other Expression Verifier Routines

The syntax analyzer and the lexical analyzer perform the bulk of the processing involved in determining the correctness of a candidate expression. Three other routines--the variable declaration module, the error message display module, and the verifier driver--are somewhat less complex. They nevertheless merit some discussion.

The variable declaration module enables the user to specify the attributes of a variable which has not been declared previously. Moreover, it restricts the user's choice of attributes to those which are valid given the context within which the variable appears. The declaration module is invoked after the lexical analyzer and syntax analyzer have successfully processed the candidate expression. It traverses the token string, stopping at each token not already declared to obtain a declaration. At any point the user may refuse to supply a declaration and thus abandon the candidate expression.

In determining what choices of attributes are appropriate for an undeclared variable, the module relies primarily upon information collected by the syntax analyzer from the variable's context within the candidate expression. This information is recorded in the token package associated with the variable and includes the type--numeric or string--and the number of dimensions (zero for a scalar) which the variable must have.

If the type of the expression as a whole is indeterminate, then it must contain at least one variable whose type cannot be determined from its context. In this case, the declaration module has no information from the syntax analyzer upon which to base a restriction of the choice of type for the variable. There is, however, another source of information. If the calling routine has specified in advance what the type of the expression must be, this is sufficient to establish the type for any variable within the candidate expression whose type cannot be established from context. Moreover, even if the caller does not restrict the type of the expression, once the user makes a declaration for the first variable--where the choice of type is unrestricted--the type of the expression as a whole is established. Consequently any other undeclared variables for which the syntax analyzer could provide no type information now have their types established as well.

The foregoing possibilities for processing undeclared variables in an expression of indeterminate type rest on the following principle: In an expression of indeterminate type, the types assigned to undeclared variables whose types have not been established by their contexts must match the type ultimately taken on by the expression as a whole. Thus, for example, if the caller requires that an expression be string-valued,

but the user's entry is of indeterminate type, the variable declaration module must require that all undeclared variables whose types are not otherwise determined by context be of type string. The principle holds because every expression of indeterminate type fits one of the following descriptions:

1. It consists of a single undeclared scalar variable or of a reference to an element of an array which has not been declared. Any undeclared variable in the subscript list for an array will have its type established by context. Thus if an expression falls into this category it contains exactly one undeclared variable whose type cannot be established from its position in the expression. Clearly the type assigned by the user to this undeclared variable establishes the type for the expression as a whole.

2. It consists of subexpressions of indeterminate type joined by the binary operator +. Such an expression can be viewed as a series of subexpressions matching the description of paragraph 1 linked by + operators. Since the operator is associative and since it requires its two operands to be of the same general type (both numeric of both string), it follows that all undeclared variables whose type is not otherwise established by context must be assigned the same type and that this will be the type of the expression as a whole.

The happy consequence is that the variable declaration module is always able to guide the user's choice of attributes for undeclared variables in such a way that the resulting expression is certain to be correct. No additional calls to the syntax analyzer are required to confirm its validity.

The error message display routine has the responsibility for issuing a diagnostic message when an error has been encountered in a candidate expression. In producing a message, the display routine will rely on a standard text used for all occurrences of the type of error in question and often also on information which applies specifically to the current instance. The standard texts for diagnostic messages are to be stored in a disk file. Information specific to the current candidate

expression is to come from an error information record accessible to all expression verifier routines. On encountering an error, a verifier routine consults this record to determine whether an error with an equal or higher priority for announcement has already been detected. If not, the routine enters a code for the new error into the record. It may also enter information which locates the error more precisely; typically this takes the form of the positions and lengths of substrings of the user's input.

The format of the diagnostic messages depends greatly on the type of user/program interaction employed by the program generator. If the generator makes use of the full CRT screen and takes advantage of cursor addressing and rudimentary graphics capabilities, an error message from the expression verifier is likely to consist of a standard text displayed at some position on the screen with highlighting of the appropriate substrings of the user's input. If, by contrast, the program generator works in a line-oriented fashion, the error message will consist of a standard text into which substrings of the current candidate expression are inserted. The former method is probably more convenient for the user. The question of which is to be used, however, is to be answered by the designer of the entire program generator. The decision on this point has little consequence for the expression verifier except as it influences the operation of the error display routine.

The driver routine for the verifier is quite straightforward in its operation. The driver logic is described in a very high level pseudo-code in Figure 6. Of particular importance is the fact that only two overlays--the lexical analyzer and the syntax analyzer--must be loaded from disk during the processing of a valid expression. Where there is a

```
call lexical analyzer;
call syntax analyzer;

if expression violates caller's requirements for type or for validity
  as target of read or assignment then

  set expression type code to "violates-requirements";

elseif an error (other than undeclared variable) has been detected
  then

  call error message display routine;
  set expression type code to "invalid";

else

  if there are undeclared variables in the expression then
    call variable declaration module;
  endif;

  if user has abandoned the expression then
    set expression type code to "user-abandoned";
  endif;

endif;
```

Figure 6. The Expression Verifier Driver Logic

defect--either an unrepairable error or one or more undeclared variables--a third overlay must be loaded. No procedure call which involves loading an overlay is embedded in a loop. Thus while processing an expression is likely to introduce a delay noticeable to the user, the time devoted to loading overlays is kept to the minimum one could expect, given the space constraints within which the verifier must operate.

ENDNOTES

[1]A consequence of this is that moving the program generator system to a larger machine--perhaps a 16-bit microcomputer with 128K or 256K bytes of primary storage--will not affect the generator's features significantly. The extra storage will best be used not to add more capabilities to the generator but to reduce the reliance on overlays and thus improve response time. One attractive possibility would be to let the entire expression verifier system reside permanently in primary storage, so that its processing would not be slowed at all by the overlay mechanism.

[2]It is conceivable that this alternative approach could be implemented without insisting that the lexical analyzer and syntax analyzer be fully co-resident in primary storage. This would require breaking the syntax analyzer into two segments--one which called the lexical analyzer and which therefore could not share its space in primary storage, and one which made no reference to the lexical analyzer and which therefore could share its space. The deffect in this scheme is that it requires two loads from disk to primary storage for each token in the input. This represents an exorbitant time penalty, and thus the approach, though conceivable, is not practical.

[3]This translation scheme is designed to be used in conjunction with a bottom-up parser, that is, with a parser which attempts to construct a derivation of the input string in reverse. The semantic actions are appropriate only for such a parser.

[4]The sizes of the LALR(1) parsing action tables illustrate this growth. For grammar 3.2, the table has 72 entries [3]. The corresponding table for grammar 3.1 requires 154 entries.

[5]The research supporting this project included numerous attempts to construct an LL(1) grammar for CBASIC expressions which incorporated the rules for identifier types. None was successful. Backhouse [4] discusses the problem of encoding type rules into LL(k) grammars and gives a simple example which, he notes wrily, "asserts that no programming language of any complexity can be LL." Backhouse suggests that type rules be left out of the formal definition of the syntax of expressions so that LL parsing may be used in compilers for practical languages.

[6]An ambiguous grammar sometimes may be used as a basis for a recursive descent parser if the recursive routines are written to avoid the conflicting parsing decisions which stem from the ambiguity.

[7]The term "predictive parsing" is sometimes used with a more general sense to include both recursive descent and the table-driven method described here.

[8]A grammar for CBASIC expressions which is suitable for use with an operator precedence parser will include a production of the form:

<factor> ::= id (<expression>, <expression>, <expression>)

The implies that that the precedence relation , = , obtains. This means, in turn, that if the topmost terminal on the stack is a comma and the current input symbol is also a comma, the parser is to shift the current input onto the stack. The parser cannot count commas--it shifts no matter how many commas are already on the stack. And since nonterminals do not enter into parsing decisions, the parser is indifferent as to the presence of an [expression] between commas.

The condition is easily detected. It is indicated whenever the parser finds that a handle beginning with "id (", ending with")", and containing at least two commas matches none of the right-hand sides of the production rules of the grammar. Recovery is also straightforward--the parser simply makes the reduction to <factor>. Determining the nature of the error precisely enough for diagnostic purposes, however, requires considerable testing, since the problem could involve one or more of the following:

1. The argument or subscript list contains more than two commas.

2. The argument list--in the form in which it appears in the handle--contains fewer than three <expression>s.

3. In addition to one or both of the above, there may be a semantic problem (for example id might be a scalar variable).

A sizeable amount of code is needed to isolate the problem so that a reasonable error message is issued.

[9]Such an error would have an intermediate priority for announcement in the event of multiple errors.

[10]Care must be taken to purge these entries from the symbol table if the user decides not to make a declaration for the corresponding names or in the event that an unrepairable error in the candidate expression is encountered.

CHAPTER IV

SUMMARY AND RECOMMENDATIONS FOR FURTHER STUDY

Summary

An expression verifier in an interactive program generator is responsible for determining whether a candidate expression supplied by the generator's user conforms to the generator's rules for forming expressions. This study has considered some of the issues facing the designer of an expression verifier for a microcomputer-based program generator intended for users who are not experienced programmers. Such a verifier must be capable of responding gracefully to errors in expressions but must make only modest demands on the microcomputer's limited primary storage space.

The error handling facilities of the verifier must provide, at the minimum, for diagnostic messages which accurately diagnose any defects in an expression and which isolate those portions of an expression where the errors lie. In addition, the verifier should incorporate a coherent scheme for signalling multiple errors in an expression; a system in which the errors are announced one at a time, with the most obscure being the first to be brought to the user's attention, is well-suited to the needs of the user and admits of a practical implementation. Finally, the verifier should permit the user to provide declarations for undeclared variables which appear in an otherwise valid expression. The

verifier should deduce the attributes of such variables wherever their contexts in the expression permit and should use this information to constrain the user's options in making declarations.

In order to reduce the verifier's requirements for primary storage space, the verifier system must make use of the target machine's overlay mechanism. This can be accomplished by dividing the verifier's processing into two phases. The first, lexical analysis, identifies each of the basic components of the expression (that is, its identifiers, constants, operators, and "punctuation" symbols) and emits a package of information for each of these tokens. The second phase, syntax analysis, parses the string of tokens produced by lexical analysis. The lexical analyzer and syntax analyzer normally reside on disk; each is called into primary storage, as needed, by the verifier's driver routine. The two modules are assigned to the same region of primary storage. The error-processing modules are also disk-resident and are loaded into primary storage only as they are needed.

The lexical analyzer can be based upon a finite state automaton which recognizes the various elementary entities of the expression language. The syntax analyzer can be built around a parser for the expression language. The designer of the syntax analyzer must resolve two issues. First, there is the question of how to test that the operands in the expression are of the appropriate data types. The type rules for the expression language could be encoded into a grammar for the language. Alternatively, the syntax analyzer might invoke semantic routines to test for compliance with the type rules as each subexpression is successfully parsed. Second, the designer must select from among the many parsing techniques now available. For the present verifier,

a syntax analyzer using semantic routines for type checking and based upon an LR parser is a reasonable choice.

Recommendations for Further Study

This project included no attempt to implement an expression verifier. An implementation would represent a reasonable extension of the work presented here. It would involve, among other tasks, the following:

1. the design of error messages for the various classes of errors, following the guidelines presented in Chapter II;
2. the construction of error recovery subroutines in the lexical and syntax analyzers to support the error handling specified in Chapter II; and
3. the construction of a finite state automaton to serve as the basis for the lexical analyzer.

The expression language accepted by the verifier specified here does not allow for references to user-defined functions. Although these functions are of questionable value given their reliance on global variables, they are part of the CBASIC language and might be supported by a program generator system. They pose a problem for the expression verifier because their argument lists may consist of an arbitrary number of expressions of arbitrary type. Although the outline of solution to this problem is not difficult to devise, implementing it efficiently is matter for further investigation.

Perhaps the most important area of further study suggested by the work presented here involves empirical tests of alternative solutions to problems posed by the user interface. The present study reasoned from experience to settle questions about how the verifier would respond to errors. Quantitative, empirical investigations might suggest better answers to these questions, especially in two cases.

1. When a candidate expression contains more than a single error, how are these errors to be reported to the user?

2. When the verifier discovers an undeclared variable, should it offer the user the option of changing the variable name (to correct a misspelling, for example) as well as the option of supplying a declaration?

The difficulty with empirical tests is that one must select some criterion on the basis of which to compare alternative solutions. For error-handling mechanisms, this choice is difficult because there are several plausible criteria. One might judge the effectiveness of such a mechanism by counting how many attempts the user must make, on average, in order to make a successful correction. Alternatively, one might measure how frequently a user abandons a candidate expression the face of some error; this would identify schemes which are confusing or intimidating. Or, again, one might evaluate the effectiveness of an error-handling scheme as a teaching mechanism by measuring how quickly and to what extent novice users learn to avoid the error the scheme handles. This list of possibilities could be extended almost indefinitely. Thus, if empirical studies of the expression verifier's user interface are to be undertaken, they must be accompanied by serious reflection about what constitutes a "good" interactive error-handling technique.

REFERENCES CITED

[1] Aho, A. V. Language theory in compiler design. In Yeh, R. T., ed., Applied Computation Theory: Analysis, Design, Modeling. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1976, 185-249.

[2] Aho, A. V. and Ullman, J. D. The Theory of Parsing, Translation, and Compiling, Vol. 1: Parsing. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1972.

[3] Aho, A. V. and Ullman, J. D. Principles of Compiler Design. Addison-Wesley Publishing Company, Reading, Mass., 1977.

[4] Backhouse, R. C. Syntax of Programming Languages: Theory and Practice. Prentice-Hall International, Englewood Cliffs, N.J., 1979.

[5] Balzer, R. Transformational implementation: an example. IEEE Transactions on Software Engineering, SE-7, 1 (Jan. 1981), 3-14.

[6] Compiler Systems, Inc. CBASIC: A Reference Manual. Compiler Systems, Inc., Sierra Madre, Ca., 1980.

[7] Conway, R. W. and Wilcox, T. R. Design and implementation of a diagnostic compiler for PL/I. Communications of the ACM, 16, 3 (Mar. 1973), 169-179.

[8] Denning, P. J. Smart editors. Communications of the ACM 24, 8 (Aug. 1981), 491-493.

[9] Green, C. The design of the PSI program synthesis system. In Proceedings of the Second International Conference on Software Engineering. IEEE Computer Society, Long Beach, Calif., and Association for Computing Machinery, New York, N.Y., 1976, 4-18.

[10] Green, C. The PSI program synthesis system--an abstract. In Proceedings of the 1978 National Computer Conference. AFIPS Press, Montvale, N.J., 1978, 673-674.

[11] Heidorn, G. E. Automatic programming through natural language dialogues: a survey. IBM Journal of Research and Development, 20, 4 (July 1976), 302-313.

[12] Johnson, R. C. Automated software development eliminates application programming. Electronics,55, 2 (June 2, 1982), 129-140.

[13] Kuck, D. J. The Structure of Computers and Computations, Vol. 1. John Wiley & Sons, New York, N.Y., 1978.

[14] Lerner, E. J. Automatic programming. IEEE Spectrum, 19, 8 (Aug. 1982), 28-33.

[15] Manna, Z. and Waldinger, R. Studies in Automatic Programming Logic. Elsevier North-Holland, Inc., New York, N.Y., 1977.

[16] Manna, Z. and Waldinger, R. Synthesis: dreams ==> programs. IEEE Transactions on Software Engineering, SE-5, 4 (July 1979), 294-328.

[17] Manna, Z. and Waldinger, R. A deductive approach to program synthesis. ACM Transactions on Programming Languages and Systems, 2, 1 (Jan. 1980), 90-124.

[18] Prywes, N. S. Automatic generation of computer programs. In Rubinoff, M. and Yovits, M. C., eds., Advances in Computers, Vol. 16. Academic Press, New York, N.Y., 1977, 57-125.

[19] Prywes, N. S. Pnueli, A., and Shastry, S. Use of a nonprocedural specification language and associated program generator in software development. ACM Transactions on Programming Languages and Systems, 1, 2 (Oct. 1979), 196-217.

[20] Pyster, A. B. Compiler Design and Construction. Van Nostrand Reinhold Company, New York, N.Y., 1980.

[21] Raphael, B. The Thinking Computer: Mind Inside Matter. W. H. Freeman and Company, San Francisco, Ca., 1976.

[22] Roth, R. L. Program generators and their effect on programmer productivity. In Proceedings of the 1982 National Computer Conference. AFIPS Press, Arlington, Va., 1982, 351-358.

[23] Shneiderman, B. Human factors experiments in designing interactive systems. Computer, 12, 12 (Dec. 1979), 9-19.

[24] Teitelbaum, T. and Reps, T. The Cornell Program Synthesizer: a syntax-directed programming environment. Communications of the ACM, 24, 9 (Sept. 1981), 563-573.

[25] Teitelbaum, T. Reps, T., and Horwitz, S. The why and wherefore of the Cornell Program Synthesizer. ACM SIGPLAN Notices, 16, 6 (June 1981), 8-16.

[26] Weizenbaum, J. Computer Power and Human Reason: From Judgment to Calculation. W. H. Freeman and Company, San Francisco, Ca., 1976.

APPENDIX A

A CONTEXT-FREE GRAMMAR FOR CBASIC EXPRESSIONS

The following are the production rules for an unambiguous grammar which defines the syntax of CBASIC expressions. It is an LALR(1) grammar which encodes the rules for precedence and associativity of operators but which does not take account of the rules governing the types of variables and constants. In the notation employed here as throughout the appendices, nonterminal symbols of the grammar are enclosed in pointed brackets (<,>), and terminal symbols more than one character in length are underlined. Multiple-character terminal symbols typically stand for a class of CBASIC keywords or operator symbols. These classes--which in a typical expression verifier would be recognized by the lexical analyzer--are defined below.

```
<expression> ::= <expression> orop <logfact>
                |<logfact>

<logfact> ::= <logfact> and <logprim>
             |<logprim>

<logprim> ::= not <logelt>
             |<logelt>

<logelt> ::= <simple expresson> relop <simple expression>
            |<simple expression>

<simple expression> ::= <simple expression> + <term>
                       |<simple expression> - <term>
                       |sign <term>
                       |<term>

<term> ::= <term > mulop <factor>
          |<factor>
```

```
<factor> ::= <element>^<factor>
           |<element>

<element> ::= (<expression>)
            | id
            | id (<expression>)
            | id (<expression>,<expression>)
            | id (<expression>,<expression>,<expression>)
            | constant
```

In the foregoing, the terminal symbol "+" represents the binary addition or concatenation operator, and "-" represents the binary subtraction operator. The terminals orop, and, not, relop, sign, and mulop stand for the following classes of CBASIC operators:

```
orop:   OR , XOR
and:    AND
not:    NOT
relop:  LT , LE , EQ , GE , GT , NE ,
        <,<=, =, >=, >,<>
sign:   unary -, unary +
mulop:  *, /
```

The terminal symbol id stands for any identifier, be it a user-supplied variable name or the name of a CBASIC built-in function. The terminal symbol constant stands for any CBASIC constant, integer, real, or string. Finally, the terminal symbols "^", "(", ")", and "," stand for themselves; that is, they are symbols which may appear in CBASIC expressions.

Note that the unary and binary versions of "+" and "-" may be distinguished easily be a lexical analyzer using one-token look-behind. If the previous token represents an identifier, constant, or closing parenthesis, a "+" or "-" is binary; otherwise, a "+" or "-" is unary.

APPENDIX B

AN LL(1) GRAMMAR FOR CBASIC EXPRESSIONS

The following are the production rules for an LL(1) grammar which defines the syntax of CBASIC expressions. Like the grammar of Appendix A, this grammar encodes the rules for precedence and associativity of operators but not those governing the types of identifiers and constants. The notational conventions and the classification of terminal symbols are the same as for Appendix A, with the additional convention that the symbol null stands for the null string.

```
<expr> ::= <logfact><expr more>

<expr more> ::= orop <logfact><expr more>
              |null

<logfact> ::= and <logprim><logfact more>
            |null

<logprim> ::= not <logelt>
            |<logelt>

<logelt> ::= <simple expr><logelt more>

<logelt more> ::= relop <logelt more>
                |null

<simple expr> ::= sign <term><simple expr more>
                |<term><simple expr more>

<simple expr more> ::= + <term><simple expr more>
                     |- <term><simple expr more>
                     |null

<term >::= <factor><term more>

<term more >::= mulop <factor><term more>
              |null
```

```
<factor> ::= <element><factor more>

<factor more> ::= ^ <element><factor more>
                | null

<element> ::= (<expr>)
            constant
            id <element more>

<element more> ::= (<sublist>)
                 | null

<sublist> ::= <expr><sublist more>

<sublist> ::= ,<expr><sublist more>
            | null
```

APPENDIX C

AN AMBIGUOUS GRAMMAR FOR CBASIC EXPRESSIONS

The following are the production rules for a grammar for CBASIC expressions. The grammar encodes the precedence rules for the unary operators (signs and the logical negation operator) and establishes the precedence of the binary arithmetic operators over the logical operators. It does not encode the rules for associativity of binary operators, nor does it establish the precedence hierarchy within the classes of arithmetic and logical operators. The grammar is suitable for use with an LALR(1) parser if the conflicting parsing table entries detected by the LALR table-building algorithm are resolved in such a way that the correct precedence and associativity rules are established. The notational conventions are those of Appendix A, with some of the terminal symbols changed.

```
<expression> ::= <expression> logop <expression>
                |not <relation>
                |<relation>

<relation> ::= <simple expression> relop <simple expression>
              |<simple expression>

<simple expression> ::= sign <element>
                       |<element>

<element> ::= id
             |id(<expression>)
             |id(<expression>,<expression>)
             |id(<expression>,<expression>,<expression>)
             |constant
             |<element> + <element>
             |<element> arithop <element>
             |(<expression>)
```

In the foregoing, the terminal symbol "+" represents the binary addition or concatenation operation. The terminals logop, not, relop, sign, and arithop represent the following classes of CBASIC operators:

logop:	OR , XOR , AND
not:	NOT
relop:	LT , LE , EQ , GE , GT , NE , <,< =, =, >=, > , <>
sign:	unary +, unary -
arithop:	binary -, *, /,^

Other terminal symbols have the same interpretation as in Appendix A. The method for distinguishing between the binary and unary versions of "+" and "-" given in Appendix A applies here as well.

APPENDIX D

A SYNTAX-DIRECTED TRANSLATION SCHEME FOR VERIFYING CBASIC EXPRESSIONS

The syntax analyzer for the expression verifier calls semantic action routines to check the types of operands in expressions, establish the types of undeclared variables when the context permits, and to determine if the candidate expression is valid as the target of an assignment or a read operation. The syntax-directed translation scheme described here is based upon the grammar of Appendix C. The notational conventions for the production rules are unchanged. The semantic routines associated with a production are sketched in a pseudo-code form. The pseudo-code has a syntax similar to that of many procedure-oriented programming languages. In lists of formal parameters or actual parameters, a symbol of the form <nonterminal> or terminal represents a data aggregate holding all of the information stored on the parser stack for the corresponding grammar symbol; in addition to the symbol itself, this includes the fields TYPE, FIRST, and LAST.

```
<expression>ø ::= <expression>1 logop <expression>2
        { call PROCESS_BINARY_SUBEXPRESSION
             (logop,<expression>1,<expression>2,<expression>ø); }
<expression> ::= not <relation>
        { call PROCESS_UNARY_SUBEXPRESSION
             (not,<relation>,<expression>); }
```

```
<expression> ::= <relation>

        {call PROCESS_SINGLE_PRODUCTION
             (<relation>,<expression>);}

<relation> ::= <simple expression>1 relop <simple expression>2

        {call PROCESS_BINARY_SUBEXPRESSION
             (relop,<simple expression>1,<simple expression>2,
              <relation>);}

<relation>::= <simple expression>

        {call PROCESS_SINGLE_PRODUCTION
             (<simple expression>,<relation>);}

<simple expression> ::= sign <element>

        {call PROCESS_UNARY_EXPRESSION
             (sign,<element>,<simple expression>);}

<simple expression> ::= <element>

        {call PROCESS_SINGLE_PRODUCTION
             (<element>, <simple expression>);}

<element> ::= id

        {call PROCESS_IDENTIFIER
             (id,Ø,id.FIRST,id.LAST,----,----,----,<element>);}

<element> ::= id(<expression>)

        {call PROCESS_IDENTIFIER
             (id,1,id.FIRST,).LAST,<expression>,----,----,<element>);}

<element> ::= id(<expression>1,<expression>2)

        {call PROCESS_IDENTIFIER
             (id,2,id.FIRST,).LAST,<expression>1,<expression>2,
              ----,<element>);}

<element> ::= id(<expression>1,<expression>2,<expression>3

        {call PROCESS_IDENTIFIER
             (id,3,id.FIRST,).LAST,<expression>1,<expression>2,
             <expression>3,<element>);}

<element> ::= constant

        {call PROCESS_SINGLE_PRODUCTION
             (constant,<element>);}
```

```
<element>ø ::= <element>1 + <element>2
        {call PROCESS_BINARY_SUBEXPRESSION
             (+,<element>1,<element>2,<element>ø);}

<element>ø ::= <element>1 arithop <element>2
        {call PROCESS_BINARY_SUBEXPRESSION
             (arithop,<element>1,<element>2,<element>ø);}

<element> ::= (<expression>)
        {<element>.TYPE := <expression>.TYPE;
         <element>.FIRST := (.FIRST;
         <element>.LAST := ).LAST;
         VALID_AS_TARGET := false;}
```

The bulk of the semantic processing is performed by the routines described below. Each routine performs the processing required by one class of productions.

```
PROCESS_BINARY_SUBEXPRESSION:
  procedure (operator,<subexpr1>,<subexpr2>,<result>);

if operator = "arithop" or operator = "logop" then

   if either <subexpr1>.TYPE or <subexpr2>.TYPE is "string" then
     set up error message; /* String found where numeric required */
   endif;

   for each of <subexpr1> and <subexpr2> do;
     if the type of the subexpression is "indeterminate" then
       set types for all undeclared variables in the subexpression
       whose types have not already been established to "must be
       numeric";
     endif;
   endfor;

<result>.TYPE := "numeric"

else /* operator = "+" or operator = "relop" */

  if both <subexpr1>.TYPE and <subexpr2>.TYPE are "indeterminate" then
    <result>.TYPE := "indeterminate";

  elseif only one of <subexpr1>.TYPE and <subexpr2>.TYPE is
  "indeterminate" then
    <result>.TYPE := (type of the subexpression for which TYPE is
                      determinate);
    set types for all undeclared variables (for which the type has
    not previously been established) in the subexpression of indeter-
    minate type to "must be (type of the subexpression for which TYPE
    is determinate)";
```

```
  elseif <subexpr1>.TYPE ≠ <subexpr2>.TYPE then
    <result>.TYPE := "indeterminate";
    set up error message; /* Mixed mode */

  else /* <subexpr1>.TYPE = <subexpr2>.TYPE */
    if operator = "relop" then
      <result>.TYPE := "numeric";
    else /* operator = "+" */
      <result>.TYPE := <subexpr1>.TYPE
    endif;

  endif;

endif;

 result .FIRST :=  subexpr1 .FIRST;
 result .LAST :=  subexpr2 .LAST;

VALID_AS_TARGET := false;

return;

end PROCESS_BINARY_SUBEXPRESSION;


PROCESS_UNARY_SUBEXPRESSION:
  procedure (operator,<subexpr>,<result>);

if <subexpr>.TYPE = "string" then
  set up error message;  /* String found where numeric required */

elseif <subexpr>.TYPE = "indeterminate" then
  set types for all undeclared variables in the subexpression whose
  types have not already been established to "must be numeric";

endif;

<result>.TYPE := "numeric";
<result>.FIRST := operator.FIRST;
<result>.LAST := <subexpr>.LAST;

VALID_AS_TARGET := false;

return;

end PROCESS_UNARY_SUBEXPRESSION;


PROCESS_SINGLE_PRODUCTION:
  procedure (<subexpr>,<result>);

<result>.TYPE := <subexpr>.TYPE:
<result>.FIRST := <subexpr>.FIRST;
<result>.LAST := <subexpr>.LAST;
```

```
return;

end PROCESS_SINGLE_PRODUCTION;


PROCESS_IDENTIFIER:
  procedure (ident,elements,first_token,last_token,<subexpr1>,
             <subexpr2>,<subexpr3>,<result>);

if ident.TYPE = "undeclared" then
  ident.ELEMENTS := elements;
endif;

if elements ≠ number of subscripts/arguments expected for ident then
  set up error message; /* Wrong number of subscripts or arguments */

else

  if elements > Ø then

    for each subscript/argument subexpression do;

      if the subexpression is of indeterminate type then
        set TYPE for subexpression to "(type required by ident)";
        set types for all undeclared variables in the subexpression
        whose types have not already been established to "(type
        required by ident)";

      elseif the subexpression is not of the type required by ident
      then
        set up error message; /* Subscript/argument of wrong type */
      endif;

    endfor;

  endif;

endif;

<result>.TYPE := if ident is a function then
                   type returned by function
                 else
                   ident.TYPE;
<result>.FIRST := first_token;
<result>.LAST := last_token;

VALID_AS_TARGET := true;

return;

end PROCESS_IDENTIFIER;
```

VITA

John Frederic Lucas

Candidate for the Degree of

Master of Science

Thesis: AN EXPRESSION VERIFIER FOR AN INTERACTIVE PROGRAM GENERATOR

Major Field: Computing and Information Science

Biographical:

Personal Data: Born in Palo Alto, California, February 14, 1955, the son of John P. and Marilyn J. Lucas.

Education: Graduated from Cortez High School, Phoenix, Arizona, in June, 1971; received Bachelor of Arts degree in Chemistry and in Religion from Claremont McKenna College in 1975; received Master of Arts degree in Religion from Claremont Graduate School in 1980; completed requirements for the Master of Science degree at Oklahoma State University in December, 1982.

Professional Experience: Research assistant, Claremont Graduate School, 1978-79; research assistant, Center for Process Studies, School of Theology at Claremont, 1979-80; graduate teaching assistant, Department of Computing and Information Science, Oklahoma State University, 1980-81; research assistant, Time Management Software Incorporated, 1981-82.