GRAMMAR-BASED FILE STRUCTURE

by

Brian William Terry

B.Sc.(Hons.), Simon Fraser University, 1985

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the School

of

Computing Science

SIMON FRASER UNIVERSITY

APPROVAL

Name: Brian William Terry

Degree: Master of Science

Title of thesis: Grammar-Based File Structure

Examining Committee:

Chairman: Dr. Wo-Shun Luk

Dr. Robert D. Cameron
Senior Supervisor

Dr. Louis J. Hafer

Dr. M. Stella Atkins
External Examiner

Date Approved: NOVEMBER 13, 1987

Title of Thesis/~~Project~~/E~~xtended Essa~~y

Grammar-Based File Structure

Author: [illegible]
(signature)

Brian W. Terry
(name)

Nov 23, 1987
(date)

ABSTRACT

A grammar-based specification of logical file structure is described. Logical file structure is the structure that the contents of a file appear to have from the user's point of view. Physical file structure is the way that the file is actually organized. Separating the two is useful because the details of physical storage are irrelevant to the user view, serving only to complicate matters, and because physical storage must be protected from misuse.

A conceptual model of a grammar-based file system is introduced and a prototype implementation is explored. The programmer specifies the logical structure with a context free grammar. The required interface routines (constructors, recognizers, selectors, parser, unparser (pretty-printer) and structured file I/O) are generated automatically from that grammar. Data structures are manipulated by means of this generated interface and can be written to and read from files, in structured form, to avoid the cost of unparsing and parsing.

The research is aimed at demonstrating the feasibility, applicability and productivity of this grammar-based approach. Specifically, we have constructed a prototype grammar-based system and have used it in an example application.

ACKNOWLEDGEMENTS

My senior supervisor, Rob Cameron, provided me with inspiration, support and guidance. Thanks, Rob. I also thank my examining committee, Lou Hafer and Stella Atkins, for their input and criticism of this work.

One of the great pleasures of graduate school is the people that one meets. I am indebted to several people in the computing department. Two in particular need to be acknowledged. Discussions with Mike Dyck helped me to understand parsers, and almost everything else of significance in GRAFS. Both Mike Dyck and Ed Merks were kind (and masochistic) enough to proofread this thesis. They will both make great senior supervisors someday.

Several people had no direct input into this thesis work but had a great deal to do with my remaining functional. In particular, Jon Forsberg and Peter Mulhern were friends when I needed them.

DEDICATION

To the two ladies who made it all possible, my wife Sylvia, and my mother Gladys.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

This thesis presents a study of the application of grammar-based techniques to the specification, implementation and instantiation of logical file structure. We begin with a discussion of abstract data types and then apply this concept to the storage of structured information in files. The result is the idea of a file data type and a system to support the use of such a type. This is followed by some reasons motivating the research into this class of system and a discussion of the research work. An outline of the thesis structure concludes the chapter.

1.1 Abstract Data Types

A data type is a set of the values and the operations that may be used to manipulate those values [GheJaz82]. Abstract data types (ADTs) are types that define values and operations independently of implementation issues. An instance of a type is an object whose value is in the value domain of that type and may be manipulated by the operations valid for that type. For example, consider the distinction between the type **INTEGER**, which is not a manipulable object, and a variable of type integer, which can be manipulated (or operated upon). It is probably more correct to consider type as an attribute of a variable rather than a variable as an instance of a type since in languages that allow dynamic binding or are weakly typed, the type of a variable is not necessarily constant. However, for the purposes of this thesis we can view a variable's type as unchanging and the variable as an instance of that type.

The intention of ADTs is to separate the essential characteristics of a type from the details of its implementation. For example, consider a list type. We want to be able to apply retrieve, add, remove, etc., operations to a list. As long as these operations are possible, correct, and expedient, then the techniques used to implement the list are not immediately important. Thus, ADTs separate a concept from its implementation.

Creating and using an ADT consists of the steps: specifying the abstract structure, specifying the required operations, implementing the required support routines, creating instances of the data type and inserting and retrieving information. A typical ADT specification is in a natural language (e.g. English). The operations are designed and implemented in an ad-hoc manner as the need for them arises. The set of operations may not be complete and may not provide a consistent interface unless the programmer is thorough. The notion of completeness is a relative one, as a set of operations is complete with respect to some property. An interface is consistent if it is possible for a programmer to predict the syntax and semantics of an element of the interface (a particular subroutine for example) from a knowledge of the conventions used to create that interface.

1.2 File Data Types

Files store information and may be thought of as an ADT [PeSi83]. The user view of file structure is usually different from the physical file structure. We say "usually" because in some cases (notably UNIX) the file structure that is seen by the user may not be much different from the underlying structure. The details of mapping the user view to the physical reality are handled by the file system. It is this abstract to concrete mapping that we are investigating.

The user view of a file that is provided by a file system is sometimes referred to as the logical file structure. In the context of this thesis, we prefer to think of this structure as *a* logical file structure, rather than the only one. The reason is that the logical file structure that is provided by the file system is, typically, different from the the logical structure that the programmer wishes that the file system had provided. In other words, a file may have several levels of logical structure, depending upon who is doing the structuring. This situation is highlighted by pointer-linked data structures. If the programmer wants to perform I/O on such entities, he is forced to construct special routines on top of those provided for basic I/O.

This thesis considers the use of context free grammars (CFGs) to specify the logical file structures that users see. Given a specific syntactic structure for files, determined by such a CFG, a set of operations on the files is defined as an ADT. This ADT is called a File Data Type (FDT). We will refer to instances of FDTs as structured files.

CFGs are well-suited to the specification of logical file structure for two reasons. First of all, they allow for the definition of files of arbitrary complexity. For example, using a CFG for a given programming language, an FDT can be set up for the programs of that language. Such programs typically have a very complex logical structure which cannot otherwise be easily dealt with.

The second reason for using CFGs in defining FDTs is that they can provide a human-readable form for the structured data. This is, of course, essential for the FDTs which are defined for the input and output of application programs. Furthermore, programmers can benefit from using such FDTs for storing and retrieving intermediate data structures as well; the ability to view and edit such intermediate files can be quite useful.

Let us consider how to support FDTs. We need to consider the following questions. First, how can FDTs be specified? Second, how can information be inserted into and retrieved from an instance of some FDT? Third, what is the scope and lifetime of the FDT and the information that its instances contain? We will deal with these questions in that order in the following paragraphs.

Before considering how to specify FDTs, let us look at what we want to specify. Programming languages provide data description mechanisms that are composed of two kinds of entities. The first kind are primitive types such as integer, character and real. The second kind are aggregation mechanisms such as structures, arrays, and unions for static data structure construction, and pointers to handle dynamic data structure construction. As these are the usual mechanisms available in a general purpose language for the construction of FDTs (and they appear to be sufficient), they are the ones that we wish to specify. This is expanded upon in Chapter 4.

What kinds of support routines are required? The answer lies in an investigation of the kinds of things that we want to do with an FDT and its instances. These things can be classified as internal (memory resident) operations or as input/output (I/O) operations.

Internal operations are those that are primarily concerned with the manipulation of instances of FDTs. The creation of a specific instance of an FDT, the selection of a particular component of an instance, and the determination of whether or not a given instance is of a particular type, are internal operations that are peculiar to a specific FDT. These operators (subroutines in the implementation) are called constructors, selectors, and predicates. Constructors are used to create instances of FDT components. Selectors are used to select specific components of instances. Predicates are used to check the type of FDT components. Some editing operations (such as replacement, deletion, and insertion of components) are examples of internal operations that can be generic, that is, not specific to any particular FDT. In a grammar-based system, this collection of operators has the properties that any syntactic element (syntagm) of the language generated by the source grammar, can be constructed, and efficient run-time checks can ensure that only correct syntactic structures can be constructed [CamIto84].

I/O operations transfer information into and out of structured files. Implicit in these transfers is the conversion of the data from one representation to another. For example, parsing a character stream representation to produce a parse tree representation. The reverse process consists of taking the instance and converting it into a format that is required elsewhere. It is extremely useful to produce a human-readable output of the contents of structured file.

We have considered, in general terms, the kinds of support and manipulation facilities that might be required. Can they be generated automatically? That is one of the questions that this research addresses.

During the discussion of support routines, we saw how information might be inserted into and retrieved from FDT instances. That is, through the offices of the internal manipulation routines or via the I/O operations. The precise details of how this is to be effected are be described in later chapters.

The third question that we want to discuss is that of the scope and lifetime of an FDT and its instances. The scope of an FDT is where it may be used and its instances may be accessed. Lifetime is the period of time that a particular FDT and its instances are available to be accessed. One of the purposes of this research is to consider the storage of structured information. Stored information can have a lifetime that is greater than the execution time of any one program or set of programs. Files can also have such lifetimes.

Since this structure is grammar-based and files are used for storage, we are discussing grammar-based file structure and grammar-based file systems (GRAFS).

1.3 Motivation for this Research

The primary inspiration and motivation for the use of grammar-based specifications is GRAMPS (GRAmmar-based MetaProgramming Scheme) [Camlto84]. GRAMPS stands for GRAmmar-based MetaProgramming Scheme. It has been used to create metaprogramming systems for both Pascal and Modula2 [Cam87b].

Several other researchers have investigated systems that are oriented around structured information. Donzeau-Gouge et al. [Don83, Don84a, Don84b], have developed MENTOR. a system for handling structured documents. MENTOR uses a sorted algebra to specify structure and a specialized language, MENTOL, to manipulate that structure. Teitelbaum and Reps have investigated syntax-based programming environments and implemented a system called the Cornell Program Synthesizer [TeiReps81]. Lamb has looked at description techniques for sharing data representations between programs [Lamb87]. This has been implemented in a system called IDL which appears to have reached the level of performance required for production use.

In the previous section, the comment was made that we needed a notation that would allow us to specify FDTs and to facilitate the automatic generation of the support routines and tables. Why would we want this? There are several reasons.

First of all, FDTs are useful from a software engineering standpoint, but are somewhat tedious to implement completely and consistently. If support routines are generated automatically then programmers will likely be more inclined to use them. The automatic generation of support routines has a number of related benefits. Automatic generation of code results in less implementation effort. Less programmer time spent on implementation of FDTs results in greater productivity, both in terms of an increased amount of programmer output and in a decreased number of machine cycles spent implementing each application. Increased use of FDTs should also have benefits with regard to program maintainability and reusability. The FDT specification is the real source of the support routines. If it is changed then the updated support routines are generated automatically with no chance of missing something important. FDTs can provide a complete and consistent interface that may be usable in several different projects.

The ability to store information in structured form avoids the necessity for the lexical analysis and parsing of data each time it is needed. Lexical analysis and parsing are expensive and as much as 50% [He86] of program execution time can be taken up by lexical analysis. There is also an increased potential for storage compaction [Cam86].

A grammar-based FDT system has the utility of a swiss army knife. Many applications fall into the input-process-output category. They are either concerned with the manipulation of instances of one FDT or with transformations from one FDT to another. It is not the intention to suggest that such a FDT support system would be the greatest thing since sliced bread, however, the uses to which it might be put seem limited only by the imagination.

1.4 Description of the Research

The intention of this research is to consider the feasibility of constructing GRAFS and the applicability and productivity of this approach. As is perhaps obvious from the above, the research has two stages. First, we implemented a prototype GRAFS, and second, we used it to evaluate the capabilities and limitations of this kind of system.

The implementation of GRAFS consists of a generator and a subprogramming system interface. The generator takes an FDT specification as input, and produces lexical analyzer tables, parser tables, and selector, constructor and predicate subroutines. The FDT specification is a context free grammar. Programmers then write application programs using both the interface specific to each FDT, and the general GRAFS interface. A detailed description of the GRAFS system and its use is given in Chapter 4 and Appendix A. An example GRAFS application appears in Chapter 5 and Appendix B.

1.5 Thesis Structure

Chapter 2 contains a discussion of the body of work related to this thesis, shows how previous work has motivated GRAFS and discusses how GRAFS differs from that previous work. It also shows how GRAFS research relates to various fields of computing science.

A detailed discussion of the research and the conceptual model of GRAFS is given in Chapter 3. We present a general model of how GRAFS-style systems can be structured and show how the prototype implementation fits into this model.

Chapter 4 discusses the GRAFS formalism and implementation. The design of the GRAFS metagrammar is discussed and the syntax and associated semantics of the major GRAFS grammar rules are introduced. The GRAFS prototype implementation is dissected and each part is discussed with respect to design details, considerations and decisions.

An example application of GRAFS and an evaluation of that example are given in Chapter 5. Finally, Chapter 6 concludes this thesis with a discussion of what has been learned from this implementation of GRAFS and some directions for further work.

CHAPTER 2

A DISCUSSION OF RELATED WORK

Give a man an inch and he wants a foot, give a man a foot and he wants a yard, give a man a yard and he wants a swimming pool installed in it. - Alfred E. Newman.

This chapter is intended to acquaint the reader with both where this research fits in the realm of computing science, and what other related work has been done. The chapter is divided into two parts.

The first part contains a discussion of the routes that one might take into this research. This area lies at the boundary of several areas of computing science. Therefore it is possible to find one's way into this work from several different directions, each with its own motivations.

The second part contains a survey of related work. It provides the reader with a background for this research, and shows how other work motivates this thesis.

2.1 An Overview of the Field

This section begins with a map of the area in which GRAFS lies. We will discuss some of the areas of computing science from which a researcher might find his way into the area of this thesis.

2.1.1 Databases and Data Dictionaries

Database work is partially concerned with separating the information contained in a database from the way in which that information is stored. The idea is that the person that is interested in retrieving information need not be concerned with the details of the location of information or the format in which it is stored. In the same vein, programs interfaced to a database should not have to be modified just because the database administrator decides, for one reason or another, that the internal organization of the database must be modified [Go84].

The need for independence of data from physical storage has lead to a quest for notations, interfaces and structures that support the separation of the structure as it appears to the user and the structure as it is actually implemented. That is, the separation of the design and implementation of the physical database from that of the logical structure seen by the user. Providing users with a notation for logical structure and a consistent interface integrated with that notation gives a basis upon which to construct database systems. Given that the interface remains constant, the implementer or system administrator is free to make whatever internal changes that he deems useful. It is just this kind of notation and interface that GRAFS is intended to provide.

Database theory and research is concerned with the relationships between data, and the independence of those relationships from the way that the data is stored. GRAFS is oriented towards individual files and the abstractions they represent. A set of access routines (constructors, predicates and selectors) is provided by GRAFS, but the purpose of the logical structure is beyond the scope and the interest of the GRAFS system. A database system might well be implemented using GRAFS routines at the lower levels.

It may sometimes be preferable to work with a grammar description of a data object rather than using a relational model of that object. For example, consider a class of entities that are naturally described by grammars, programming languages. It is very difficult to separate the notion of grammars from the description of the structure of programming languages. Thus it may be more natural to use a grammar-based system directly for those objects whose structure is normally described by a grammar. Some work has been done by Linton [Lin84] using a relational database to store programs. Interestingly enough, the conversion of the program text into relations was done by a parser and not by the database system. There is also the problem of modeling various language aspects using relations. Consider the problem of dealing with nesting. If we define a domain of subroutines, then the problem of describing the nesting relationships of those subroutines may be complex, whereas this relationship is quite naturally expressed by a grammar.

Data dictionaries are somewhat similar in that they maintain, in a centralized location, information about the data structures used in and between systems. The idea is to have a consistent structural definition used in all of the programs and systems that make use of that structure [Go84]. This may include consistency in the routines used to manipulate the data structures, in which case access routines are either generated automatically by the system from each structure specification or are hand-coded by a programmer and kept inviolate. Automatic generation implies less work for the programmer and a correct and up-to-date implementation. The more complex the structure that can be dealt with, the less work for the programmer. These are some of the issues that GRAFS addresses. A data dictionary can use GRAFS for construction of some of its physical storage routines. GRAFS could be used to implement low-level database operations but is not intended to replace high-level interfaces.

2.1.2 Programming Environments

Programming environments are complex software tools used to assist in the programming process [BaShSa84]. The intention is to integrate the various tools used during the programming process (editors, compilers, profilers, etc.) in such a way as to streamline and enhance the programming process [HeeKli85]. Environments can range from integrated interactive and interpreted language environments for APL or Lisp, to ad hoc collections of software tools for Fortran or Pascal.

One thing that is common to all kinds of programming environments is the necessity for dealing with data structures; both the program source code, and the data objects that programs manipulate. There are two aspects of the current research in programming environments that are of interest with respect to GRAFS. First is the automatic generation of parsing, unparsing and structure-access routines. This capability makes it less work to create these environments. Second is the ability to store structured data easily. This allows the information in the data structures to persist between program invocations.

2.1.3 Abstract Data Types

As previously mentioned, ADTs are desirable. Thus anyone interested in research that concerns ADTs and associated methodologies or using ADTs in production work will find themselves considering abstract structure specification and automatic code generation sooner or later.

Research in this area addresses issues ranging from software engineering issues concerning the interaction of ADTs with the software life cycle, to mathematical issues of modeling ADTs as algebras. Although there seems to be some conviction that "many sorted algebras are the right mathematical tools to explain what abstract data types are." [EhMa85, p. 2], they do not seem appropriate for writing down specifications of larger systems [EhMa85, p. 3]. Hence there is a need to look for specification languages for larger systems. Such a search, in combination with more specialized specification requirements, could lead to grammar-based approaches like GRAFS.

2.1.4 Metaprogramming

Metaprogramming is the activity of writing programs that have programs as data objects. This covers quite a wide range; anything from the ubiquitous text-editor to a compiler. However, in order for a metaprogram to perform operations of any significance, it needs to know about the structure of the programs being manipulated. Thus we confine the definition of a metaprogram to one that has knowledge of program structure.

Analyzing the structure of a program involves the syntactic analysis of that input program. It should be noted that this is not the only way to construct metaprogramming systems, but it is a natural one. The metaprogram works with a program's parse-tree structure rather than its textual representation. If the metaprogram performs any transformations, then the transformed program can be converted back into textual form by unparsing the parse-tree representation. These are a set of operations on an ADT, the ADT in this case being the parse tree type.

Viewing a metaprogramming system as an ADT facility that is specialized to syntactic structures leads to GRAFS. There are many different programming languages and hence there exists the problem of creating metaprogramming systems for each language. Since these systems can be grammar-based, a researcher in this area might find himself looking at GRAFS-like systems with the idea of automatically generating metaprogramming systems for different languages.

2.2 Related Work

In this section, we consider directly applicable or similar work that has been done as well as some further motivation for GRAFS. We want to show the elements of current research that support this thesis, and to differentiate this research from that done by others.

2.2.1 Historical

This sub-section considers the facilities that have been traditionally provided for FDT support. These are mostly limited to programming language I/O facilities and operating system file subsystem interfaces.

The I/O facilities of existing programming languages (eg. PL/I) allow the file record structure to be specified [PolSte80]. However, the access routines must in general be constructed by hand since the built-in I/O routines will not handle dynamic pointer-linked data structures directly. Thus, changes in the file or record structure will, in general, result in changes being made in several places, making maintenance difficult and error prone. Any data abstraction must be implemented and enforced manually.

Consider again the notion of type applied to files. If a system is unaware of the real structure of a file, then it has no way of ensuring that access is compatible with that structure. In this instance, file type is not strong: files have a specific logical structure but can be accessed without reference to or with incorrect reference to that structure. This weak typing allows bugs to go undetected.

Operating system access methods are primarily concerned with machine performance. OS's can use several predefined methods for accessing files, and sometimes the programmer is given the choice of which one to use [PeSi83]. However, this choice is not altogether useful for the logical definition of the file structure since it concerns only efficient access to file records. Thus, no matter which access method is chosen, the programmer must define the logical file structure and hand code the appropriate abstract access routines.

The fact that the programmer must specify the logical structure of the file, regardless of the OS support, suggests the existence of two dimensions for classifying file systems. The first dimension is the variation of access routines for maximum efficiency in accessing individual bytes in the file. The second dimension is the variation in the manner in which the system supports the definition of the logical structure of the file. This thesis research lies in this second dimension.

2.2.2 Recent Work

The last section was concerned with what sorts of logical file structure support facilities are generally available to most programmers. This section is concerned with current research in this area. It is interesting to look at the way in which GRAFS meshes with other research. Other work may have similar intentions but be different in approach, or similar in approach but with different or more specific intentions (e.g., GRAMPS).

Some motivation for this research and confidence in its usefulness and success derives from the work of Cameron and Ito [CamIto84]. Their application of grammar-based techniques to the specification and automatic generation of metaprogramming systems (GRAMPS) has been quite successful. GRAMPS has been used to implement systems for both Pascal and Modula2 [Cam87b]. GRAFS uses similar grammar-based techniques for the more general task of data structure and file structure specification. However, GRAFS is not concerned with providing higher-level functionality. Such things as semantic analyzers, structure editors and specialized scanning routines are beyond the immediate scope of GRAFS, although they might be constructed using GRAFS. The aim of the present research may be taken as both a subset and a generalization of their work. GRAFS is a subset in the sense that it does not provide all of the programming language oriented support routines of GRAMPS and a generalization in that GRAFS is intended to apply to other applications as well as metaprogramming.

Some related work has been done by Donzeau-Gouge in a system called Mentor [Don83], [Don84a] and [Don84b]. Mentor is concerned with the organization of documents such as programs, prose (papers, books, etc.) and specifications. A BNF grammar in their earlier work [Don83], and an algebraic specification formalism in their more recent work [Don84b] are compiled to generate a parser, unparser and access functions. The difference between GRAFS and Mentor lies in the scope of applicability and the efficiency of the implementation. In Mentor, a specification-independent language, called MENTOL, is used to manipulate the abstract tree structure. Thus, GRAFS and Mentor have very different interfaces in that the GRAFS interface is intended to reflect the data structure being manipulated, whereas MENTOL is independent. GRAFS is intended to be used directly on top of, or perhaps in place of, OS file system facilities in an existing programming language. A system like Mentor could then be implemented using the lower-level structured file support provided by GRAFS. GRAFS is intended to create an efficient general-purpose tool that can be used to construct higher-level tools and systems such as Mentor.

Another approach has been taken by Lamb [Lamb87], with a system called IDL (Interface Definition Language). This work is concerned with using a notation to define data structure interfaces between different programs. The notation involved is similar to a grammar but is used in stages. The rough structure is defined in one specification and then other specifications are used to supply various structural and semantic refinements. IDL attempts to provide maximum flexibility of semantics and hence it requires more programmer input in the code generation process than does GRAFS. Also, IDL is not as strongly typed as GRAFS since IDL allows the programmer a good deal more latitude in the manner that data structures are manipulated. IDL is very much concerned with transformations between data structures and as such is similar to GRAFS. Its implementation seems to have passed the raw prototype stage and is almost ready for production use [Lamb87].

Another problem that could lead to GRAFS is data transfer between different machines. Sun XDR (eXternal Data Representation) [Sun85] is a protocol for the representation and transfer of data. This work is quite similar to that of Herlihy and Liskov [HerLis82], although the Sun document contains no references to that work (or any other). Different computer systems have different internal representations for data, so moving data from one machine to another requires converting to the representation required on the target machine. Transferring data around n different machines requires n(n-1) (2 for each pair of machines) conversion procedures. If we make use of a standard representation then we need only construct 2n different conversion procedures; namely an encoder and a decoder for each machine. This considerably reduces the cost of program construction and maintenance, at least as far as conversion programs are concerned.

The implementation of XDR consists of a set of routines for encoding and decoding primitive C language data objects, such as ints, floats, etc.. Aggregate objects such as structures, require the user to construct custom encoder/decoder routines on top of the primitive XDR routines. To transfer data, the user opens an XDR data stream, packs his encoded data into it, and decodes the data at the other end with a corresponding decoder on another machine. The stream is not intended for manipulation other than sequential reading and writing, since it is data encoded as a byte stream, making editing operations (such as delete and replace) difficult. XDR does not keep track of where a particular data object is, leaving it up to the user. The implementation of the encoding and decoding routines is left to the user; nothing is generated automatically, so any change in the data object specification requires a manual change in the encoding/decoding subroutines.

XDR and GRAFS have different intentions. XDR is concerned with physical representation while GRAFS is concerned with logical representation. GRAFS could use XDR to implement the internal file I/O routines.

Specialized environments make strong use of ADTs. For example, Lisp environments are oriented around lists and MPS is oriented around abstract parse trees. It is convenient to store information between runs of the programs that use it, thus allowing several different programs to be invoked with the same information. This sort of facility is available in environments for Lisp and APL. While these facilities must generally be custom built for each application, with GRAFS they can be generated automatically.

CHAPTER 3

THE GRAFS CONCEPTUAL MODEL

This chapter introduces a conceptual model of GRAFS and contains a more detailed discussion of the research. The first section contains a discussion of the derivation and structure of a conceptual model of GRAFS. The second section deals with the expressive power of GRAFS. That is, what kinds of structures is it possible to construct using GRAFS. The third section discusses the nature of the research and what it attempts to show. We consider the potential benefits of such a system. The evaluation of the actual benefits is left to chapters 5 and 6.

3.1 The GRAFS Conceptual Model

What is the GRAFS conceptual model? It is a description of what this sort of system might look like, and why it might look that way. We begin with the idea of storing structured information in files. As was noted during the discussion of database research in Chapter 2, files have both a logical structure and a physical structure. For several reasons, we would like to keep these separate. ADT methodologies are intended to separate logical structure from implementation details (physical structure). We apply ADT concepts to the task of storing structured information in files, to get the concept of FDT. This is the kind of thing that programmers do all the time, but any time people are free to (or have to) build as they will, they are also free to make errors. Thus, we would like to specify FDTs, and have them enforced by a structured file system.

What kinds of structures is the file system going to know about and allow? The same kinds of structures that are available in current programming languages, since those are the structures that programmers use. The more the file system knows about these structures, the more support that it can give. This suggests providing the file system with a specification of each structure. Also, since we want to have the file system control access, it has to have control of the access routines. This means the file system must have either a fixed, specification-independent interface, or else an interface that is custom-built for each specification. Regardless of which is chosen, it is necessary to refer back to the specification. In the case of the independent interface, the routines must check with the specification (in one way or another) to see if the action requested is allowed in the given structural context. Specification-specific routines would have to do something similar, but because they are compiled from the specification rather than having to interpret that specification, there should be some performance advantage. Also, compiling the specifications to give a specific interface could result more readable code, if the names of the interface routines reflect the structure that they manipulate.

Grammar-based techniques provide a specification notation, and the ability to easily generate a syntactically complete set of access routines. Furthermore, if we limit the grammar to being context free, then any structure manipulation operations are local in scope. If the system views the structure as context-free, then changes are local and do not propagate throughout or depend upon the whole of the structure. This is important from a performance perspective.

The GRAFS conceptual model is diagrammed in Figures 1 and 2. This model has two main components. The first part is the GRAFS program interface consisting of the subroutines that application programs use to communicate with GRAFS. The second is the generator which takes an FDT specification and produces the required support routines and tables.

GRAFS is essentially an FDT support system. As mentioned in Chapter 1, we require the means to instantiate, manipulate, and perform I/O on typed files. Instantiation and manipulation are, at least conceptually, fairly straightforward. Constructors, predicates, and selectors are complete, in a grammar-based approach, in the sense that they allow the creation and manipulation of any and all sentences of a language. However, I/O is a bit more complex. We must deal with the questions of which data is to be stored, where it is to be stored, and how it is to be stored. The first question involves what kinds of objects I/O must handle. The second question

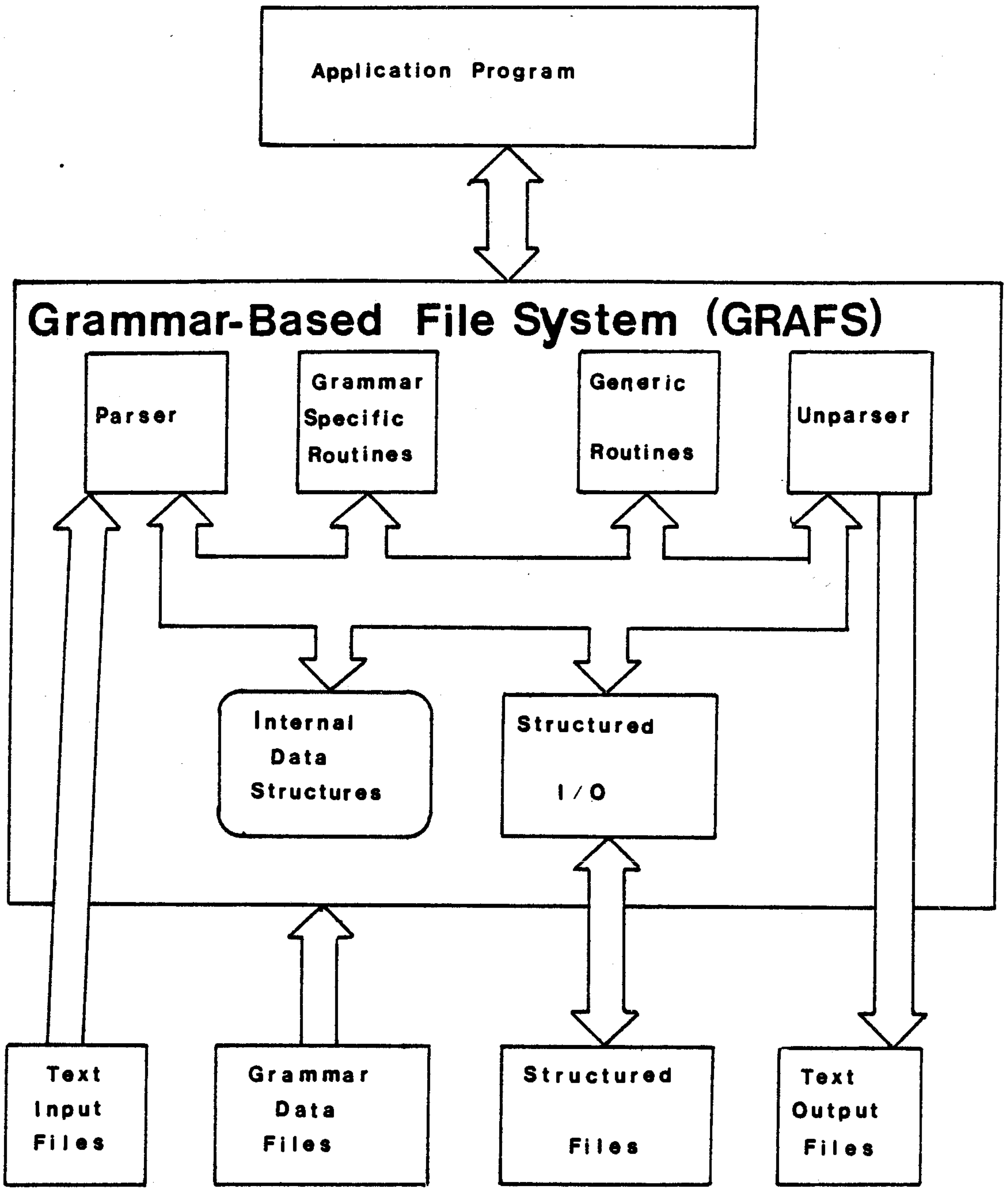

Figure 1: The GRAFS System Structure.

involves the source and destination of I/O operations. The third considers the ways in which data can be stored, either as text or in structured form.

I/O comes in two flavours, between GRAFS and the outside world, and within GRAFS itself. One of the motivations for this research is the desire to store structured information in a file, without the need for programmer intervention (i.e. hand-built access methods). Thus we need an I/O subsystem that handles structured GRAFS files.

The other I/O requirement is the need to get data into GRAFS from outside GRAFS and, from time to time, write it back out. One motivation for this is that people often find it useful to see the contents of their data structures. A typical situation that arises is the necessity for data structure dumps when debugging programs. It would be convenient to have dump routines provided automatically. Also, a lot of information is stored as only implicitly structured text and so it would be convenient to convert text streams into GRAFS structured files automatically.

In general, data that is external to GRAFS might be stored in any fashion and thus might require customized conversion facilities. The reason for this is that the structure to be input might exceed the recognition capability of the parser. In these cases, conversion routines might be constructed using the internal manipulation routines to construct a GRAFS structure directly under the direction of a hand-built parser. For example, consider generating a metaprogramming system for the C programming language. The C grammar is difficult to express using a context free grammar, and is therefore not amenable to a context free parse.

The last component of GRAFS, which is not visible to the user, is the grammar-data module. Each separate GRAFS grammar needs to have its own set of records that are to be made available to the rest of the system as required. The prettyprinter, and some generic access routines, have to check into details of a particular grammar before being able to perform their functions.

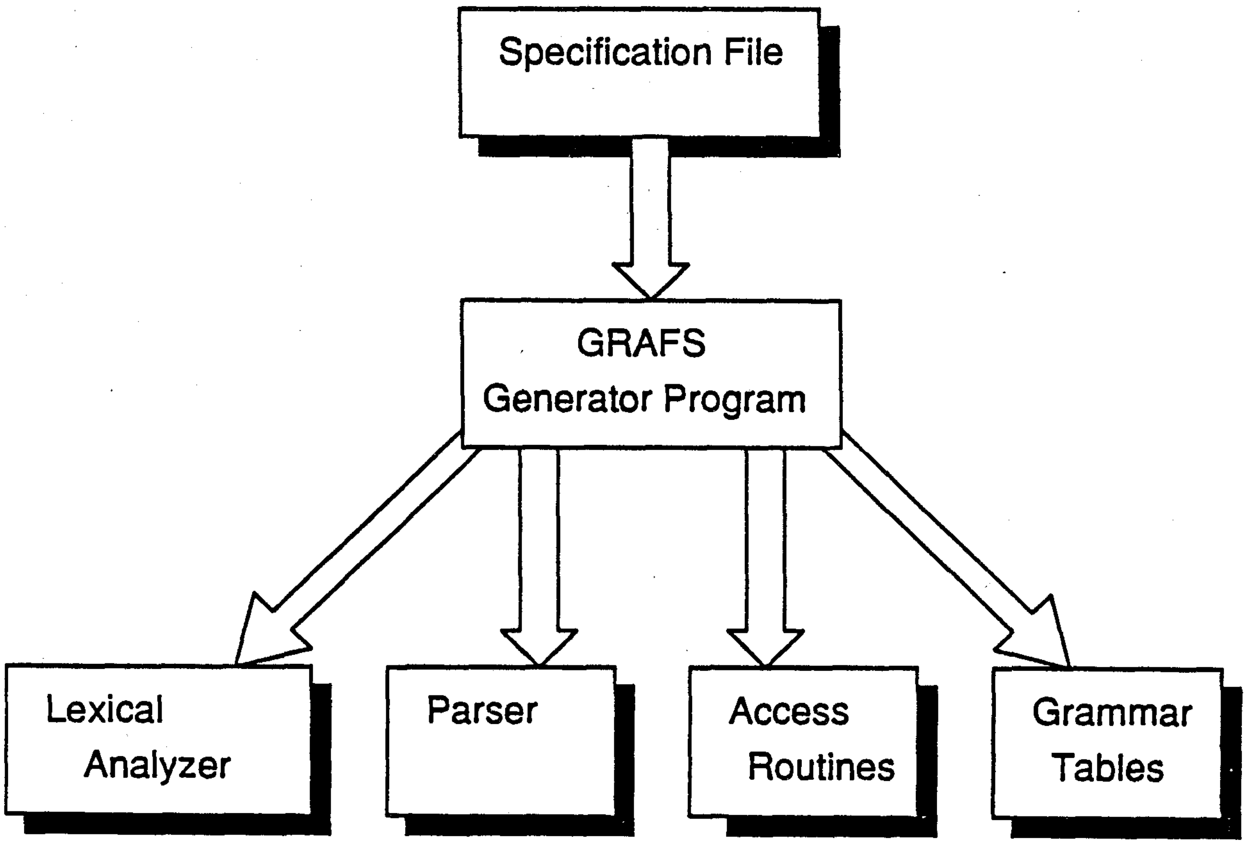

Figure 2: The GRAFS Generator Structure.

The parts of the program interface are diagrammed in Figure 1. The old file system interface should probably be kept available, through GRAFS, to maintain compatability with other systems. Even if GRAFS turns out to be the wave of the future, untyped files will continue to exist for some time, thus GRAFS will have to retain the ability to handle them.

The design of the generator is driven by the requirements of the program interface. We require a recognizer (parser and lexical analyzer), grammar-specific manipulation routines, various grammar tables, and what ever other grammar-specific entities are needed in the implementation. The generator model is shown in Figure 2.

The generator can be thought of as a compiler of GRAFS specifications. It reads the specification, performs lexical analysis and parsing operations on it to produce a parse tree. That parse tree is then used as input to a parser generator, a lexical analyzer generator, an access routine generator and whatever other routines that are required. The resulting data is written to files in a format known to the rest of the GRAFS system.

3.2 Expressive Power

The details of the GRAFS notation and its associated semantics will be dealt with in Chapter 4. However, something that we might consider at this point is the expressive power of a grammar with respect to FDTs. Herlihy and Liskov [HerLis82] state that the naming relationship among objects can be modeled as a directed graph. The situation where one object that references another object by name, can be represented graphically as two nodes that are joined by a directed arc. This suggests that graphs are useful for modeling data structures. Therefore, we can consider what sorts of graphs can be constructed using a GRAFS grammar.

We can certainly construct trees of arbitrary depth and complexity. Nodes of any degree can be constructed using the GRAFS construction and repetition operations. A particular structure may be repeated arbitrarily many times via the grammar mechanisms of recursion or repetition. GRAFS lets you construct arbitrary trees automatically, but arbitrary graphs containing cycles must be handled differently. In other words, entities such as circular lists have to be supported by the application program that is using GRAFS.

Let us consider cyclic data structures in more detail. How could we add cycles to GRAFS structures? There is no provision for self-reference in a grammar in the sense of referring to a particular instance via, for example, a pointer. A specification is of structural type, it cannot know which particular nodes will be used in a structure of that type. This sort of referential information must be added when the structure is actually created. However, it is still possible for the application to create cycles. Depending upon how GRAFS is implemented, it may be possible to use editing operations on the parse tree to introduce cycles. If the implementation does not allow this then virtual cycles can be created. The application can keep track of extra links between nodes in a separate structure. These extra links are not handled by GRAFS, are not seen by it, and therefore create no linearization difficulties during I/O operations. There is also the option of using labels. A particular field of a record could act as the label (or name) of that record and other records could then refer to that particular record by referencing its label.

It is possible to model arbitrary graphs with GRAFS, however this just covers the form of structures, it does not handle their content. We now have a method for creating arbitrary graphs, but what kinds of nodes and edges can we use? The aggregation operations of construction, alternation, and repetition, should allow the construction of any desired structure. Leaf nodes are the entities that correspond to integers, reals, characters, etc. These are known as the base types of a programming language. Leaf nodes in GRAFS must be able to handle arbitrary base types. This can be handled in the same way that programming languages do, by providing a selection of base types. Thus, GRAFS can have the same ability to describe data structures as its host language.

3.3 Direction of Research

The research consisted of implementing a rudimentary GRAFS based on the conceptual model just described, and using it in an example application. The implemented system allows the programmer to specify the logical structure of files and then generates the required support routines automatically. The logical structure is specified using a context free grammar (CFG). The required access routines, parser tables (if required) and unparser (prettyprinter) tables are generated automatically from that specification.

The research is intended to address questions of the feasibility, productivity and applicability of this approach. Feasiblity is demonstrated by the successful construction of a prototype system. Productivity is considered in the context of the computer, programmer and user. Applicability is shown by using the prototype system in an example.

Computer productivity derives from the effective utilization of machine resources. There are several ways that GRAFS effects this. GRAFS allows information to be stored in structured form thus avoiding unnecessary conversion overhead (parsing and unparsing) and facilitating data compaction. Lexical analysis and parsing are expensive [He86] so we want to minimize the number of times that these operations are performed. Instead of parsing the input and creating new data structures each time a program is run, that input can be parsed once and the resulting data structures stored in a file between runs. As we noted in Chapter 2, an example of this lies in the use of metaprogramming systems such as MPS [Cam87b]. If the internal parse tree representation is stored in a file, then parsing need only be done once, thereby increasing the efficiency of the metaprogram. In fact, if the parse tree is constructed directly, as with a structure editor or with constructor routines, then parsing need not be done at all. Editing operations can be done directly on the parse tree and text output can be produced when required by unparsing the tree.

Programmer productivity stems from the effective utilization of programmer resources. This is enhanced by GRAFS in the following ways. First, the programmer need no longer spend time writing and debugging structure-specific code. Second, the fact that the specification is the source of the data structure and support routines should serve to decrease maintenance activities because changes are made in one place (the specification) and support routines are generated automatically. Third, FDTs are desirable from a software engineering standpoint, since the automatic generation of access routines will likely motivate programmers to use an FDT methodology.

User productivity comes from the effective utilization of the user's resources. As this is influenced by computer and programmer productivity, the improvements cascade. The increase in programmer productivity shortens the time necessary to fill user requests. Improved computer utilization results in better system response and increased resource availability to users. Greater reliablility from stronger file typing results in fewer bugs and happier users.

Applicability is shown by using the GRAFS prototype to implement an example application in a conceptually cleaner manner and with less effort than would be required if the GRAFS system were not used. Specifically, the example application is a re-implementation of the lexical analysis sub-system of GRAFS. We had to construct a lexical analyzer generator and interpreter for the GRAFS prototype, and the example is a re-implementation of that sub-system using GRAFS.

It is impossible to prove (in a rigorous sense) any of these properties about GRAFS, since they are all rather subjective and it would be difficult to perform experiments that would give conclusive results. It is therefore hoped that the implementation and application serve to demonstrate the potential of GRAFS-type systems. The intention of this research is to show that a GRAFS-type system is feasible and that this class of system merits further investigation.

As might be expected, the GRAFS prototype is aimed at functionality rather than optimality. In other words, we were more concerned with building a working prototype than in constructing a production quality system. With this in mind, we used existing facilities when possible and constructed basic facilities when needed. For example, we made use of the Modula-2 file system interface in a very rudimentary fashion. The result was a functional but far from optimal GRAFS prototype.

CHAPTER 4

GRAFS FORMALISM AND IMPLEMENTATION

This chapter deals with the GRAFS formalism and implementation. It is divided into two sections, the first dealing with aspects of the formalism and the second with implementation details.

A simple example can help to clarify aspects of this material. Therefore, we will consider the construction of a data structure for a simple personal address book, using GRAFS. The idea is to create a data structure to hold a list of address records. Each of these records contains a person's name, address, phone number, and occupation. We will use this example to illustrate various facits of the GRAFS formalism and implementation. The complete grammar for this example is given in Figure 17.

4.1 GRAFS Formalism

The GRAFS formalism is modeled on that of GRAMPS [CamIto84]. GRAMPS, as has been previously noted, is a grammar-based methodology for specifying and generating metaprogramming systems.

4.1.1 Designing a MetaGrammar

GRAFS has to be told about the structure of a given FDT. In this section, we develop a notation to describe FDTs. We first consider the nature of the grammar specification rules.

GRAMPS uses four different kinds of rules: construction, alternation, repetition and lexical, although lexical rules are not strictly part of the GRAMPS formalism [CamIto84]. The GRAMPS grammar notation itself is an extended Backus-Naur Form. Implementations of GRAMPS-style systems (e.g., Pascal MPS, [Cam87b]), are partially generated and partially hand-coded.

GRAMPS metaprogramming systems are not often constructed, whereas GRAFS specification grammars are to be constructed often, and by programmers of varying experience. Thus, the issues of providing a relatively firm, easy to understand and completely defined notation do not arise in GRAMPS, but do arise in GRAFS.

A general problem in grammar design that also arises in GRAFS is of how much of the GRAFS specification grammar is to be dealt with as a recognition problem and how much is to be dealt with by semantic analysis. For example, in the GRAFS formalism, alternation rules are composed of only non-terminals. This restriction is enforced by the GRAFS metagrammar, rather than by internal checks. The metagrammar is constructed such that only non-terminals are syntactically correct in an alternation rule. In some cases it is very difficult to specify restrictions as a recognition problem. Consider the case of using a grammar to specify only those numbers between 59 and 898. It is possible to do, but not easily or concisely [Pag81]. In this instance it is better to treat the problem semantically. Why create extra parser or lexical states when a single pair of comparisons will tell you whether the candidate number meets your criteria? Also, why complicate the logical structure of the target grammar when the only result is to make programming more difficult? If such restrictions are made part of the GRAFS grammar itself then offending specifications cannot be expressed as a well-formed GRAFS specification. This does not mean that the user cannot make mistakes in the specification, but at least the GRAFS grammar specification can serve as a reasonably complete documentation of what is correct and what is not. Thus, there is a tradeoff between syntax and semantics, and some judgement must be exercised in deciding on the best approach to take.

The GRAFS grammar specification is itself a grammar. It is a metagrammar: a grammar grammar, that is also its own grammar. The GRAFS metagrammar describes all well-formed GRAFS grammars including itself. The self-description provides an interesting bootstrapping opportunity as we shall see in the implementation section of this chapter.

Designing the GRAFS metagrammar involved deciding on a notation and then expressing that notation in terms of itself. This is a rather challenging activity since it involved expressing an incomplete notation in an incomplete notation as the notation is being developed. Not only must a notation be gradually developed to express GRAFS grammars but it must also be extended to express itself. This leads almost at the very beginning to problems with the use/mention distinction. The problem comes in many guises and it cropped up often during the design and development of the GRAFS system. This problem manifests as the mental confusion that results when one forgets to distinguish between whether the notation is describing itself or something else. That is, one slips mentally from one level to another or fails to distinguish use from mention. It is not within the scope of this thesis to deal with the psychological issues inherent in the use/mention distinction, however as it was a factor in the design and implementation it is included as part of the experience.

The next issue that arose in the search for a notation was that of associated semantics. The GRAFS system is intended to perform certain actions based upon the grammar handed to it. Since the abstract structure of GRAFS data objects must be known to the user, the semantics of the GRAFS grammar must also be known to him. For this reason, it was decided that constructs that had different semantics should have different notation.

As noted by Cameron and Ito [CamIto84], the structure of the grammar used in grammar-based systems is rather more critical that in the case of more monolithic applications (e.g., compilers). The structure of the grammar should reflect the data structures that will be obtained from it. Thus, we required a parsing algorithm that was capable of easily creating the structures described by the GRAFS notation. For these reasons an ELR(1) (Extended LR with a 1 token lookahead) [PurBro81] grammar was chosen. In ELR grammars, the right hand side of a production is a regular expression. Thus each ELR grammar rule can have concatenation, repetition and alternation operators in it. However, the actions that GRAFS takes for, and the structures created by, each of these operators are quite different, so for reasons of clarity, each of these operations occurs as a separate GRAFS rule class. Thus, the GRAFS grammar rules are a subset of those of ELR. The result of all of this is that the GRAFS grammar rules can be parsed as is; no transformations need be done on the grammar. More will be said about this in Section 4.2.3.

What are the rule classes of the GRAFS notation? They are the construction, alternation, repetition, lexical, and lexical-class rule classes. The first three classes are aggregators and roughly correspond to the record, union and array constructs, respectively, that are found in procedural programming languages. The repetition construct also corresponds to lists. The difference in the interpretation as an array or as a list lies in the way that an element is accessed (more on this in the next section).

Lexical and lexical-class rules are used to describe primitive data types which are the types that are aggregated into structures, unions and arrays. They are considered as the leaves or terminal nodes of the parse tree.

4.1.2 GRAFS Rules: Syntax, Semantics and Raison d'Etre.

This section discusses each rule and its associated semantics in turn. We will use structures for the address book example to illustrate each rule. The GRAFS metagrammar is shown in Appendix A.

Construction rules correspond to records. They aggregate an arbitrary but fixed number of data objects of specified types. This heterogeneous collection of objects may now be referred to by one name; the name of the construction rule that describes that particular aggregation.

The syntax of construction rules is quite simple and is shown in Figure 3. It consists of the keyword "CONSTRUCT", followed by the name of the specific rule (a simple non-terminal), followed by the keyword "IS", all followed by a list of the components of the rule. Each component may be a terminal string (keyword), a compound non-terminal (a data element), a formatting

```
<ConstructionRule> ::= ID "CONSTRUCT"
  <Name:SimpleNonTerminal> "IS"
  <ConstructionElements:ConstructionElementList> OD

<ConstructionElementList> ::=
  <ConstructionElement> { <ConstructionElement> }

<ConstructionElement> ::= <Terminal> |
  <CompoundNonTerminal> | <OptionalPhrase> | <Directive>
```

Figure 3: Construction Rule Syntax.

directive (for the prettyprinter), or an optional phrase.

Keywords are signposts for the parser and the user. They are used in general grammar design to disambiguate the grammar and document the output. For example, they are the reason that no explicit separators (periods or commas for example) are required in the grammar itself. They also serve to add contextual information that may aid the user in writing the input or reading the output. For example, consider the case where the input data is a series of identifiers, each of which has a specific meaning to GRAFS. The use of keywords can serve to remind the user of the significance of each of the otherwise possibly indistinguishable objects. Keywords have no further significance to GRAFS, and are not available or manipulable as data fields of the construction rule of which they are a part. This will become clearer when we look at the GRAFS programming interface.

Compound non-terminals correspond to field declarations in conventional programming language record types, have two functions and hence two parts. The first part is called the component name and the second is called the class name.

The component name corresponds to a record field name and is the device by which the user selects that component. The class name corresponds to the type of a record field, and is defined by some other rule of the grammar in question. Each compound non-terminal in a given construction rule must have a component name that is unique to that rule. Note that the same component name may be used in several different construction rules. Also note that the component name and the class names may be the same, on those occasions when inspiration fails.

Optional phrases were included for programming convenience. It is possible to achieve the same result by defining a separate construct for each variation but this is clumsy. In the worst case, for one option, there needs to be two separate rules, for two options, four rules, and so on. This rapidly grows annoying. Also, the resulting application programs are more cumbersome since each different rule must be checked for and dealt with separately. The use of optional phrases allows the program to merely check for the existence of the given option. If it is there use it, if it is not then do not. Optional phrases need to be used with some care to avoid introducing ambiguities into a grammar. The use of keywords can be of some assistance here. As we shall see when we get to repetition rules, lists may have zero or more elements. Thus, the user should not in general use list specifiers directly (that is, without guiding keywords) within an optional phrase. If the list is empty, the parser generator will not know what to do. Is this an empty list or a nonexistent one?

The last kind of component that can occur in a construction rule is a formatting directive. Formatting directives are special keywords that help to control the operation of the prettyprinter (there are other ways of controlling it as well, as we shall see in the implementation section of this chapter).

The choice of formatting directives is still an open question [Opp80, Rub83, Wood86], so the set of directives chosen for GRAFS consists of only those directives that were found necessary during implementation. Undoubtedly, as more projects are undertaken with this system, other directives will be required.

There are five directives currently available. They are ID, OD, LB, NTS, and TS and stand for InDent, OutDent, Line Break, No Token Spacing and Token Spacing.

ID and OD move the left margin setting in one tab stop and out one tab stop respectively. The actual value of tabs can be found or set using the GRAFS interface. The new value of the left margin does not come into effect until the next linefeed. These two directives are intended to allow the nesting of various structures, thus each ID should be balanced at some point with a corresponding OD. If this is not done, the printed output will have a slanted look to it. Possibly, future implementations could reset indents automatically upon leaving a structure.

The LB directive forces line breaks. When the prettyprinter encounters one of these in the grammar rule that describes the node currently being printed, it immediately prints an end-of-line and continues printing at the left margin of the next line.

NTS and TS control the white space between individual tokens. There may be situations where a group of tokens should be printed out in a contiguous stream with no separators between them. An example of this (and in fact, the motivation for the inclusion of these directives) may be found in the compound non-terminals in the metagrammar. These elements just did not look right printed as "`< component : class >`" and turning the spacing off gave "`<component:class>`", which looked much better. This example highlights the problem of where to place white space and where not to. The problem could be handled by the construction of a customized prettyprinter (more on this in the implementation section). However, we felt that it would be useful to deal with as many problems as possible by the use of formatting directives. The chosen solutions have thus far proven quite adequate.

Now that we have described construction rules, let us look at their use in the address book example. Figure 4 shows a construction rule that defines a single record of the address book. As was previously mentioned, the address book will consist of a list of these records. Notice the use of the LB directive to force linebreaks so that the name and address fields, are on their own lines. The `Record` is a simple structure that is, in essence, a set of keyword/value pairs. The keywords (e.g. "Name:") are used here for documentation purposes to tell the reader what each field means.

Figure 5 shows a construction rule that defines the structure of an address. Note that the `Apartment` field is optional. The reason for this is that not everyone lives in an apartment. In order to avoid confusing the parser we add the keyword "APT".

```
CONSTRUCT <Record> IS
 "Name:" <Names:Names> LB
 "Address:" <Address:Address> LB
 "Phone:" <Phone:Phone>
 "Occupation:" <Occupation:Occupation>
```

Figure 4: Construction Rule Example: Address Book Record.

```
CONSTRUCT <Address> IS
 ID [ "APT" <Apartment:Number> ]
 <StreetNumber:Number> <Street:Identifier> LB
 <Town:Identifier> <Country:Identifier> <Code:Code> OD
```

Figure 5: Construction Rule Example: Address Record.

The next GRAFS rules are alternations. The syntax of these rules is much simpler than that of construction rules and is shown in Figure 6. An alternation rule is composed of the keyword "ALTERNATE", followed by the name of the alternation rule, followed by the keyword "IS", followed by a list of non-terminals separated by vertical bars. The non-terminals used in this rule must be simple and not compound as is the case with construction rules. Simple non-terminals have only a class name, no component name and hence no ":" separator. These class names are, as before, the names of other rules in the grammar that specify other structures. The component name is not required here since no selection operation need take place.

Alternation rules correspond roughly to unions as used in C or Pascal. This rule class specifies the set of things that can be used in places where particular rule identifiers are used.

```
<AlternationRule> ::= ID "ALTERNATE"
  <Name:SimpleNonTerminal> "IS"
  <Alternatives:SimpleNonTerminalList> OD

<SimpleNonTerminalList> ::=
  <SimpleNonTerminal> { | <SimpleNonTerminal> }
```

Figure 6: Alternation Rule Syntax.

Alternation structures need not be present in the structure being built internally. Alternation rule names that occur in other rules can be thought of as place holders that may be filled by any of the alternatives given in that rule. There is no selection to be done, since there is only one element in use at any time. Also, this shortcut tends to simplify programming, since the subroutine calls necessary to pass through the alternations (and which would have no other function) are no longer required. Cameron and Ito [CamIto84] used this approach and this research has uncovered no difficulties with it.

Let us see how alternations are used in our address book example. Consider the representation of names. Since people do not all have the same number of names, we will use a list. We have the further complication that some people use initials instead of their full name. Thus, we need the option of having a name being an identifier (the name spelled out) or an initial. The alternation rule in Figure 7 allows a name to be either spelled out or an initial.

```
ALTERNATE <Name> IS <Identifier> | <Initial>
```

Figure 7: Alternation Rule Example: Name or Initial.

The last aggregation rule is repetition. Repetition structures correspond to arrays or lists, depending upon how they are accessed. That access may be direct, using the position number of the desired element, or relative, using "next" operations. Repetition structures have the advantage that the user need not be concerned with an upper limit on the number of elements, as is the usual case with arrays. Repetition structures, like arrays (and in most cases, lists) are homogeneous. The elements of the repetition are all of the same type. However, that type may be an alternation, in which case you have the effect of non-homogeneity to a limited extent.

The syntax of repetition rules is shown in Figure 8. Each rule starts with the keyword "LIST", followed as before by the name of the rule, followed by the keyword "OF", followed by the simple non-terminal designating the type of each element. Next comes the keyword "SEPARATOR", followed by the separator character to be used in this list. A blank may be used if desired. After that comes an optional list of formatting directives.

Separator characters serve the same purpose in repetition rules as keywords do in construction rules. They aid the parser in figuring out what it is looking at and they aid the user in writing the input and reading the output. For example, blanks are used in the construction element list portion of construction rules. At the moment, separators are restricted to being single characters. There is no particularly good reason for this restriction, it results from lack-of-imagination on the part of the implementor and can be easily corrected if required. There are other possibilities for separators, such as character strings or no character at all, but so far single characters have proven to be adequate. Multiple character separators will be required however, we will leave this enhancement to the next implementation.

There are three places in repetition constructs where formatting is important. These are at the beginning of the list, between each element of the list and at the end of the list. Formatting before and after can be dealt with by using formatting directives in an enclosing construction rule. Since the list is homogeneous, each element should be treated identically, thus the list separator and inter-element formatting are specified once. The formatter list given in a repetition rule takes effect between each element of the list, after the separator character is printed. If there is more than one formatter in the list (e.g. LB ID) then the prettyprinter acts on each one in turn. If there are no formatters in the list, then no action is taken. Experience has shown that no explicit

```
<RepetitionRule> ::= ID "LIST"
  <Name:SimpleNonTerminal> "OF"
  <BaseType:SimpleNonTerminal> "SEPARATOR"
  <Separator:Terminal> <Formattors:DirectiveList> OD

<DirectiveList> ::= <Directive> { <Directive> }
```

Figure 8: Repetition Rule Syntax.

formatting was required in most cases and that when it was, a simple forced line break (LB) was sufficient.

We talked about using a list of address records for our address book. Figure 9 shows a repetition rule that defines `Records` as a list of `Record` structures. The separator is blank because we are going to use blank lines between records so we do not need explicit separator characters. The blank line separation is achieved with the two LB directives. These directives cause the prettyprinter to perform two line breaks in succession between each two records in the list. The result is a blank line between each pair of records in the prettyprinted output.

```
LIST <Records> OF <Record> SEPARATOR " " LB LB
```

Figure 9: Repetition Rule Example: Address List.

That completes our discussion of aggregation rules. Now we need to have some primitive data objects to aggregate. There are two rule classes that are used to define these primitives; lexical rules and lexical-class rules. Lexical rules are the real bread-and-butter rules. The lexical-class rules are not, strictly speaking, necessary and were included for implementation reasons. Simply, lexical-class rules make it easy for GRAFS to construct simpler finite automata by letting it know that there is a transition between two states for all the members of a lexical-class. It is possible to get the same result using state-minimization algorithms, for example, but why force GRAFS to do unnecessary work.

```
<LexicalRule> ::= ID "LEXEME"
  <Name:GenericNonTerminal> "IS"
  <Definition:RegularExpression>
  [ <Delimiter:DelimiterExpression> ] OD

<DelimiterExpression> ::= ID
  "L-DELIMITER" <LDelimiter:OneCharacter>
  "R-DELIMITER" <RDelimiter:OneCharacter> OD

<RegularExpression> ::= <Term> | <AlternationExpression>

<Term> ::= <Factor> | <ConcatenationExpression>

<Factor> ::= <Terminal> | <ClosureExpression> |
  <LexicalNonTerminal> | <BrackettedExpression>

<AlternationExpression> ::= ID <Operand1:Term> "|"
  <Operand2:RegularExpression> OD

<ConcatenationExpression> ::= ID <Operand1:Term>
  <Operand2:Factor> OD

<ClosureExpression> ::= ID "{"
  <Operand:RegularExpression> "}" OD

<BrackettedExpression> ::= ID "("
  <Operand:RegularExpression> ")" OD
```

Figure 10: Lexical Rule Syntax.

Lexical rules allow the user to tell GRAFS how to recognize primitive data objects (tokens). The lexical rule name is the name of a token and the rule body describes how to recognize one. The syntax of lexical rules is shown in Figure 10. These rules begin with the keyword "LEXEME", followed by the name of the particular rule, followed by the keyword "IS". Then follows a regular expression that describes the items to be recognized. The last part of the rule is optional, and is used to specify the right and left delimiters. If this option is exercised, the keyword "L-DELIMITER" followed by the single-character left delimiter to be used, followed by the keyword "R-DELIMITER", followed by the single-character right delimiter, are added to the specific rule.

The regular expression may be constructed using alternation, concatenation and repetition operations, in any combination. These operations have precedence (highest to lowest) repetition, concatenation, and alternation. The operators are right associative because of the parser implementation used. However, bracketting is available to force the desired precedence and associativity.

The primitive objects in a regular expression are terminals and lexical non-terminals. Terminals are strings of characters delimited by double quotes ("). Lexical non-terminals are similar to simple non-terminals except that the delimiters used are pairs of "#" rather than angle brackets. They have been added as a notational convenience for the user to allow a more compact expression. For example, identifiers are one letter followed by zero or more letters or digits. Writing out the regular expression for this using single character and digit strings would be a bit tedious and would tend to obscure an otherwise straightforward concept. Lexical non-terminals are defined using other lexical rules or lexical-class rules. The reason that lexical non-terminals are distinct from simple non-terminals is to control where they may occur in the grammar. The use of simple non-terminals instead would open the door to using aggregation rules to define elements occurring within regular expressions. This could cause some implementation difficulties, as we shall see later in this section when we consider the lexical analyzer/parser interface.

Delimiters can be a problem for the lexical analyzer. Are they part of the entity that they delimit or not? The initial GRAFS lexical analyzer did not recognize delimiters as special. This lead to having to deal with them explicitly -- sometimes an annoyance. There will be more on this problem in the section on lexical analyzer implementation (since that is where the problems arose). However, for the moment consider a regular expression that begins and ends with " and one that begins and ends with "a". The first expression is supposed to be a delimited string and the second is an identifier (for example). Any time strings were dealt with, it was necessary to strip off the delimiters. In order to have this stripping done automatically, it was necessary to tell the system explicitly what a delimiter was. So we introduced the optional part of the lexical rule class. If the user does not want to have the delimiters stripped, it is only necessary to leave out the optional section.

Figure 11 shows a lexical rule that defines an `Initial` for our address book. This is a very simple regular expression that defines `Initial` as any one letter followed by a period ".". Since no delimiters are specified explicitly, the delimiters are assumed to be blanks. In other words, in order for the lexical analyzer to recognize "W." as an initial, there has to be a blank after the ".".

```
LEXEME <Initial> IS #Letter# "."
```

Figure 11: Lexical Rule Example: Initial.

Lexical-class rules were added for implementation reasons. They are used to specify a character class, for use in other regular expressions. The same thing can be accomplished using lexical rules but there are some differences in the way that the lexical analyzer tables would be constructed. More will be said about this later.

Lexical-class rule syntax is very simple and is shown in Figure 12. These rules consist of the keyword "LEXICAL-CLASS", followed by the name of the rule (a lexical non-terminal) followed by the keyword "IS", followed by a list of single characters delimited by quotes and separated by blanks. The effect is that of an alternation rule of single character strings. An example of lexical class rules, taken from the address book grammar, is shown in Figure 13.

```
<LexicalClassRule> ::= ID "LEXICAL-CLASS"
 <Name:LexicalNonTerminal> "IS" <Members:CharacterList>

<CharacterList> ::= <OneCharacter> { <OneCharacter> }
```

Figure 12: Lexical Class Rule Syntax.

Earlier in this section, the term "associated semantics" was mentioned with respect to GRAFS rules. There are two aspects of GRAFS rules, what the rule looks like and what the rule

```
LEXICAL-CLASS #UppercaseLetter# IS "A" "B" "C" "D" "E" "F"
  "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
  "R" "S" "T" "U" "V" "W" "X" "Y" "Z"

LEXICAL-CLASS #LowercaseLetter# IS "a" "b" "c" "d" "e" "f"
  "g" "h" "i" "j" "k" "l" "m" "n" "o" "p"
  "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"

LEXICAL-CLASS #Digit# IS "0" "1" "2" "3" "4" "5" "6" "7"
  "8" "9"
```

Figure 13: Lexical Class Rule Example.

means. We have discussed what the rules look like, it remains to consider what the rules mean.

Determining meaning was not much of a problem with aggregation rules. Records, unions and arrays/lists seem to be universal concepts and easily understood. However, what does a lexical rule mean? Along with use/mention, this was a major conceptual problem. The problem that we had stemmed from not understanding that the recognition and conversion of tokens to internal forms are separate processes, even though they may be intertwined. There are two parts to the process of analyzing tokens. First of all, the tokens must be recognized. Recognition is the process that is specified by regular expressions and is performed by finite automata. Recognition tells you what the thing being looked at is. The second process, conversion, is turning the character string representation into a form that the rest of the system can deal with. For example, we see an integer as a string of digits, whereas the computer sees integers as a binary encoding. Thus, in order for the rest of the system to deal with a token as an integer, the token must be converted into the form of an integer. Conversion is not as straightforward a process as recognition since it is implementation dependent. Hence, we decided to avoid the problem in the prototype GRAFS by restricting leaves to the type, character string. This form requires no conversion. If the application program using GRAFS requires that the information be in some other form, then the application program can perform the conversion. The lexical class rules for the address book example are shown in Figure 13.

4.2 GRAFS Implementation

This section discusses the GRAFS implementation. GRAFS is in the raw prototype stage. Therefore, we were mostly concerned with finding some way of getting GRAFS to work, rather than with finding the most efficient way, since it was generally not obvious (at least initially) where the bottlenecks in the system were going to be.

4.2.1 Bootstrapping

Bootstrapping is a term found in the domain of operating systems. It means having a system gradually getting itself running by itself or pulling itself up by its bootstraps. The system begins small and loads or constructs more and more of itself until the entire system is operational. Bootstrapping has a similar meaning with regards language implementation. In the case of GRAFS, it simply means that we use part of the system to construct another part of the system.

The situation is as follows. The system must take a specification and compile it to produce, amongst other things, tables for the parser and lexical analyzer. In order to do this we need to perform lexical analysis and parsing on the specification and then use a parser generator and lexical analyzer generator on the specification's parse tree, to produce tables for a parser and lexical analyzer for structures in the language of the specification grammar input.

Why not use the generator routines to produce tables for the GRAFS metagrammar and thus save the effort of hand-coding a parser and lexical analyzer? This turns out to be not terribly difficult to do, although it does create a nice conceptual loop that can cause all kinds of confusion. In order for the generator routines to work, there must exist a parse tree (or whatever structures the generators need to see in order to work). In order to get a parse tree we need a parser, or do we? No, we can pretend that we have a parser. All we need is the parse tree. So, we write a routine that constructs the parse tree in memory (using constructor routines that are custom-built for the metagrammar) and then run the generators on that parse tree to produce parser and lexical analyzer tables for the metagrammar.

The result of implementing this bootstrapping technique is a program called *boot*, which when run, produces the tables and files necessary to compile GRAFS specifications. The metagrammar evolved during the implementation phase of this thesis, necessitating modifications to boot. These changes were very easy to make, even the most involved took only three or four hours. Given the later experience with the complexity of parsers, it is unlikely that a hand-coded parser would have been easier to modify.

4.2.2 Lexical Analysis

Lexical analysis is a critical phase of compilation since it can take up to 50 percent of the total compilation time [He86]. Thus, if you are building a compiler, you will want to construct a fast lexical analyzer. Lexical analysis is not such an overwhelming issue in GRAFS, or at least not from a performance standpoint. The reason for this is that one of the purposes of GRAFS is to eliminate much of the need for lexical analysis.

Unfortunately, GRAFS needs to do lexical analysis from time to time. The generator program has to lexically analyze specifications that are input as text, and users may wish to have textual input converted into structural form. These events are seen as relatively rare, since most of the time I/O should involve only structured files, so the speed performance of the lexical analyzer is not as critical. This is fortunate since the lexical analyzers used by GRAFS are automatically generated and such analyzers tend to be slower than their hand-coded counterparts.

Heuring [He86] has done some interesting work on the generation of fast lexical analyzers. Unfortunately, his approach does not appear to be particularly flexible, in that it requires the tokens to be partitioned into classes according to how those symbols are to be recognized. This requires that the designer be able to so partition his tokens and that he be able to communicate all of this to the GRAFS generator. Since we wanted to maintain maximum flexibility and interface simplicity in the prototype, we did not pursue this approach further and instead chose the conventional table interpreter approach.

The lexical analyzer generator traverses a specification's parse tree and constructs a non-deterministic finite automaton (with epsilon moves) from the specification keywords, strings, and any regular expressions defined in lexical rules. An equivalent deterministic automaton is then constructed via the classical algorithms [AhoUll79]. The resulting tables are written to a file. That file is read in as required, and interpreted by the lexical analyzer.

4.2.3 Parsing

Earlier in this chapter, we mentioned that an ELR(1) parsing algorithm was used. In this section, we will go into a bit more detail as to why Extended LR(1) was chosen.

If GRAFS was a monolithic system, the choice of parser, providing that that parser was sufficiently powerful, would not matter. In this case, the user would remain blissfully ignorant of the nature and form of the structures created internal to the system. However, this is not the case with GRAFS. The user is expected to write programs that manipulate the internal GRAFS data structures and therefore must have a strong understanding of those structures. The structures that are actually generated by GRAFS must behave exactly as the user expects them to. If these structures are not what the user expects, manipulation will prove difficult.

GRAFS data structures are described by a grammar. Each different rule in the GRAFS grammar describes a different and distinct structure. The question now is, how do we use the grammar notation to produce a physical structure? There are two ways: with the access routines (described in the next section) or with the parser. We are going to generate both access routines and parser for each different grammar. Access routines are easy to generate, but parser generation is much more difficult.

There are a number of different parsing algorithms. We needed one with sufficient power. The obvious choice was some version of LL or LR. Which one to chose? LR is the more powerful

[AhoUll79], and so was a tempting choice. However, the existing LL and LR algorithms use a BNF grammar notation that is quite different from GRAFS grammar notation. For example, repetition is achieved by recursion in BNF, whereas, GRAFS has repetition rules. Thus, in order to use a conventional parser, we would need to transform the GRAFS grammar into a BNF. This would result in the construction of a slightly different structure than the original GRAFS grammar suggests. We could, of course, re-transform that structure to correspond with the original grammar, but it would be much nicer to avoid the problem altogether. Fortunately, there exist various ELR(k) algorithms, specifically [PurBro81], that remove most of the problem.

Extended LR is based on a slightly more powerful notation than that of BNF. Basically, the right-hand-side of each production is a regular expression, with alternation, concatenation and closure (repetition) operations. This fits very well with the GRAFS grammar notation. In fact, ELR is more expressive than the GRAFS notation since GRAFS right-hand-sides are not full-blown regular expressions. Thus, we chose to use the ELR(1) algorithm for the GRAFS parser.

4.2.4 The Parser/Lexical Analyzer Interface

An interesting design issue in GRAFS arises from the existence of two stages in the process of constructing a parse tree from a text string. These stages are lexical analysis and parsing. Lexical analysis is the process of breaking the input stream into tokens. Parsing is the processing of taking the input tokens and creating a parse tree. The interesting thing about this is that the parser is capable of handling the task of token recognition without the aid of a separate lexical analyzer. Since lexical analysis is a simpler process than syntax analysis, it is possible to construct a more specialized and efficient recognizer for tokens than for syntactic structures.

If we make the decision to split the recognition process into two stages, we face the problem of deciding how the grammar notation is to be dealt with. That is, which rules are for the parser and which are for the lexical analyzer. It might be possible to make GRAFS smart enough to figure this out on the fly, however for the prototype we decided on a fixed method. Construction, alternation, and repetition rules specify syntactic structure and are dealt with by the parser. Lexical and lexical-class rules specify tokens and are handled by the lexical analyzer.

This division of labour is fine from the point of view of GRAFS. However, what has happened is that the programmer is left to make the decision of which construct is handled by which level of analyzer. There are a number of things to be considered in making a decision.

What is the information that is to be manipulated? Consider that a simple non-terminal, and identifier delimited by angle brackets, could be treated as a single token. However, the significant information is in the identifier, not the angle brackets, so the brackets would need to be stripped by the application program. If the simple non-terminal is dealt with as three tokens, then only the identifier need be dealt with.

The other consideration is programmer convenience. Sometimes it is simply easier to specify something as a single token. This is usually because there are several similar tokens that could overlap and hence be very difficult for the lexical analyzer to recognize. Consider the case of delimited strings. As was mentioned earlier in this chapter, the string delimiters are not really part of the string that they delimit. The delimiters act as signposts for the analyzers. Once we know that something is a string, the delimiters are no longer required. The temptation might be to handle delimiters as separate tokens, but this can result in other difficulties. For example, consider the string " abc ". Are the blanks before and after the "abc" part of the string or not? This is a somewhat artificial example, in that the meaning can be decided upon. However, it seems that although delimiters are not part of the token itself, they are not really separate from the token either, and so should be handled at the same time.

The parser/lexical analyser interface solution that was chosen for the GRAFS prototype seems to work well enough. However, future such systems might investigate other alternatives.

4.2.5 Access Routines

At various times we have introduced the topic of access routines. We have mentioned the grammar-specific constructors, selectors, and predicates, and the existance of generic, or grammar-non-specific, routines. This section discusses the routines actually implemented and some of the issues of that implementation.

Access routines fall into two categories: grammar-specific and generic. Grammar-specific routines are generated for each particular grammar and only apply to that grammar. Generic routines are built into GRAFS, are not changed by GRAFS, and apply to any appropriate grammar and structure.

Grammar-specific routines fall into three classes: constructors, predicates, and selectors. The first class, constructors, is used to make structures in the grammar. Alternation structures are not needed in the parse tree so there is no point in being able to make them. Thus, "Make" routines are generated for construction, repetition, and lexical rules only. The details of the routine internals vary somewhat between rule classes, but are fairly consistent within the same class. This allows us to use code templates for each class, and simply fill in the blanks as required.

Consider the following example. The construction rule that we wish to generate a "Make" for is:

```
CONSTRUCT <REInput> IS <REList:REList> <SubREList:SubREList>
```

The resulting "Make" routine is shown in Figure 14. An examination of the routine will reveal the places where blanks needed to be filled in. For example, the name of the routine, the structure type (code), and name string, all come from the name of the rule "REInput". The name of the grammar is taken from the name of the file that contains that grammar specification, in this case "REGrammar". The number of parameters, their types and positions are taken from the rule body.

```
PROCEDURE MakeREInput ( x1, x2 : Node ) : Node ;
VAR n : Node ;
BEGIN
  n := MakeNodeRecord ( "REGrammar" ) ;
  SetClass ( n, CONSTRUCTION_RULE_CLASS ) ;
  SetCode ( n, REInput ) ;
  SetNodeType ( n, NONTERMINAL ) ;
  SetNameString ( n, 'REInput' ) ;
  GrammarCheck ( x1, "REGrammar" ) ;
  IF NOT ( REListQ ( x1 ) ) THEN
    Error ( "MakeREInput: argument 1 is not a REList." ) ;
  END ;
  InsertComponentK ( n, x1, 1 ) ;
  SetParent ( x1, n ) ;
  GrammarCheck ( x2, "REGrammar" ) ;
  IF NOT ( SubREListQ ( x2 ) ) THEN
    Error ( "MakeREInput: argument 2 is not a SubREList." ) ;
  END ;
  InsertComponentK ( n, x2, 2 ) ;
  SetParent ( x2, n ) ;
  RETURN ( n ) ;
END MakeREInput ;
```

Figure 14: An Example Modula-2 'Make' Routine.

Repetition rules generate a similar "Make" routine except that, since lists can contain zero elements, there are no parameters. Creating a list node involves first creating a nil length list using a "Make" routine, and then using the generic append operations to add elements to that list.

Lexical rules are different in that the argument to the "Make" routine is a string rather than some number of nodes. Lexical rules specify leaf nodes that contain information stored as a character string.

Predicates are generated for all rules except lexical-class rules. The name of each of these routines consists of the name of the rule used to generate it, followed by the letter "Q". These routines are functions returning a boolean value. Each routine checks the grammar type of the node argument and then checks to see if its argument node is of the same type of node that would be constructed by the grammar rule that was used to generate the routine.

The observant reader should now be asking "ah yes, but what about alternation rules?". Alternation rules are not used to generate nodes, but they do define classes of nodes. Thus, the alternation rule predicates perform a logical "or" operation on the predicates of each of the elements in the body of the alternation rule. For example, consider the following alternation rule:

```
ALTERNATE <Type> IS <Accept> | <Reject>
```

The resulting predicate is shown in Figure 15. This routine is, again, just an exercise in filling in the blanks.

```
PROCEDURE TypeQ ( x1 : Node ) : BOOLEAN ;
BEGIN
 IF x1 = Node ( NIL ) THEN RETURN ( FALSE ) ; END ;
 GrammarCheck ( x1, "FAGrammar" ) ;
 RETURN ( AcceptQ ( x1 ) OR RejectQ ( x1 ) ) ;
END TypeQ ;
```

Figure 15: An Example Alternation Rule Predicate Routine.

The final routine category is that of selectors. There are two rule classes that generate selectors: construction and lexical. Construction rules define a collection of named fields. Thus, each field can be addressed, or selected, by it symbolic name. Each field in a construction rule is a compound non-terminal. Compound non-terminals have two parts, the first being the name of the field, and the second being the type of that field. The first part is used to generate a selector routine for that field. Generation of construction rule selectors is complicated somewhat by the fact that the same field name may occur in different places in different rules, although a name can only occur once in any given rule. For example,

```
CONSTRUCT <A> IS <B:Thing1> <C:Thing2>
```

is okay, whereas,

```
CONSTRUCT <A> IS <B:Thing1> <B:Thing2>
```

is definitely NOT okay.

A particular selector name may occur in several different rules. This implies that a particular selector name will have different results depending upon the rule. If the host language does not allow overloading then we can only have one routine of that name, which will have to service several different types of nodes. Thus, each selector routine must check its node argument to see which kind of node it is before the routine can make the selection. The best way to make this clear is with yet another example. Consider the following two construction rules:

```
CONSTRUCT <Default> IS "NEXT_STATE" <NextState:Number>

CONSTRUCT <Transition> IS
 "CHARACTER" <CharacterList:CharacterList>
 "NEXT_STATE" <NextState:Number>
```

This results in the "NextStateOf" selector which is shown in Figure 16.

```
PROCEDURE NextStateOf ( x1 : Node ) : Node ;
VAR n : Node ;
BEGIN
 GrammarCheck ( x1, "FAGrammar" ) ;
 CASE CodeOf ( x1 ) OF
 Default :
  n := GetComponent ( x1, 1 ) ;
 | Transition :
  n := GetComponent ( x1, 2 ) ;
 ELSE
  Error (
   "NextStateOf: Operation incorrect on this node." ) ;
 END ;
 RETURN ( n ) ;
END NextStateOf ;
```

Figure 16: An Example Selector Routine.

Lexical node selectors are a bit different in that there is not much choice about what is to be selected. The idea is to extract the information contained in the node and return it to the caller. The only form that the information can take in the current GRAFS implementation is character string. These routines are named "Retrieve" followed by the name of the lexical rule, and then followed by "Of". So a lexical rule for "Number" would cause a "RetrieveNumberOf" routine to be generated.

We will finish off this discussion of grammar-specific routines with a look at some of the access routines that would be generated for the address book example. The construction rule `Record` in Figure 4 would have the constructor `MakeRecord`, the predicate `RecordQ` and the selectors `NamesOf`, `AddressOf`, `PhoneOf`, `OccupationOf` (corresponding to the name, address, phone and occupation fields) generated for it. The construction rule `Address` in Figure 5 would have the constructor `MakeAddress`, the predicate `AddressQ`, and the selectors `AptOf`, `StreetNumberOf`, `StreetOf`, `TownOf`, `CountryOf`, and `CodeOf` generated for it. Thus if the node `currentItem` was a `Record`, in which case `RecordQ ( currentItem )` would return true, then the street that the person:
`NameOf ( currentItem )` lives on would be:
`StreetOf ( AddressOf ( currentItem ) )`.

The alternation rule for `Name` shown in Figure 7 would only have a predicate `NameQ` generated for it. `NameQ` would be true if and only if `IdentifierQ` or `InitialQ` were true.

The repetition rule for `Records` shown in Figure 9 would have both a predicate and a constructor generated for it. The predicate would be called `RecordsQ` and would return true if its node argument was a list of records. The constructor `MakeRecords` takes no arguments and returns an empty list of records. Records may be added to the list using a generic append operation. Selection on lists is also a generic operation since every element is the same. Thus list elements must be refered to either by their position in the list, or by their relationship with some other member of the list (i.e. give me the next one after this one, or give me the previous one, and so on).

Generic access routines are constructed by the GRAFS implementor as the need for them becomes apparent. They are difficult to categorize completely for the reason that it is difficult to know, *a priori*, all of the routines that might be wanted. The word "wanted" is used intentionally, since grammar-based techniques can result in a syntactically complete set of operations (as discussed in Chapter 3). However, it may be more convenient to add routines that perform operations in a more effective manner. For example, it is useful to have routines that allow data structure navigation without being tied to any particular grammar structure. This allows the construction of generalized search and replace routines, for example, that may be used in several different situations.

What can be said about generic routines, in general, is that they must have access to specification information on the structure being manipulated. This is to ensure that the requested operation is relevant to the structure that that operation is being applied to. For example, the append operation should only be applied to repetition class node.

The GRAFS implementation is a prototype and so generic routines were constructed as required. The existing generic routines fall into the categories of parsing, prettyprinting, structured file I/O, list manipulation and generic predicates. All of these operations are abstract in that they apply to any and all grammars.

Parsers differ in the source of their input. For example, one might wish to parse text stored in a particular file, taken from a standard input, or taken from a character string. Only one version is currently implemented and that one is designed to parse the text that is stored in a particular file (the name of that file is an argument to the parser).

Prettyprinters differ in the choice of output destination and formatting routines. One version, called `PrettyPrint`, uses default formatting routines and sends its output to *stdout*. The other version, called `UnParse`, takes nothing for granted. The user provides a set of formatting routines that control when and where output occurs. More is said about this in the section on prettyprinting.

GRAFS does not automatically store its structures in files. Therefore, if the user wishes to use a previously stored structure, or store a structure for future use, GRAFS must be informed.

There are two routines provided for this purpose. `SaveParseTree` saves the specified structure (GRAFS views everything as a parse tree, hence the name), under the supplied name. `RecoverParseTree` reconstructs the structure of the supplied name, and returns the root note to the caller.

Lists are structured the same way in all grammars so they can be accessed using generic routines. There are three such routines currently available: `NthElement`, `ListLength`, and `AppendNodeToList`. `NthElement` retrieves a child node from the supplied list by number. This is similar to array accesses; you specify the element required by its ordinal number. `ListLength` returns the number of elements in the supplied list. `AppendNodeToList` glues the supplied node onto the end of the supplied list. There are many other operations (e.g. next, first, previous, insertNthElement, etc.) which could also be constructed.

Generic predicates are predicates that apply to classes of nodes. The only one currently implemented is `EmptyNodeQ`. This predicate is true if its argument exists and false otherwise. There are situations (such as optional nodes) where a nonexistant node may be selected. This predicate allows the application to test for such cases. Many other generic predicates could be constructed. For example, rule-class predicates could allow programs to scan structures in a generic manner. An example of this sort of generic scanning occurs in PascalMPS [Cam87b].

Access routines are generated in two files for each grammar. One file that contains a Modula-2 definition module that includes all of the access routine header declarations for that particular grammar. The other is a file that contains the corresponding implementation module. These files must be compiled and then linked to the user's application program. The names of the two files are taken from the name of the grammar being processed. For example, input of a grammar file named. G1, would result in the generation of the files G1_GS.def and G1_GS.mod, containing the definition module and implementation module, respectively, for that grammar.

4.2.6 PrettyPrinting

Constructing a prettyprinter is not difficult, it is mostly a matter of dealing with a number of special cases. The details of the special cases are dependent upon the choice of formatting directives and the nature of the structures that are to be printed. Each structure type and format directive requires its own special handling. In this sense, the GRAFS prettyprinter algorithms are quite standard. The classic reference on prettyprinting is Oppen [Opp80].

What criteria drove the design of the GRAFS prettyprinter? As usual there are two main approaches, hard-coding a custom prettyprinter routine, or using a table interpreter with special tables for each structure specification. For reasons of flexibility and speed of implementation, we chose the table interpreter. The grammar information is already available in parse tree form and manipulation routines were already in existence so we decided to use the parse tree directly. This meant that because the prettyprinter had to know how its tables were structured then every grammar had to be of the same type, so that one set of access routines would apply to all specification grammars. All grammar specifications (including the metagrammar itself) are instances of the metagrammar and hence have the same grammar type, "META".

Given the grammar-based nature of the specifications and the programming-language bias of prettyprinters it seemed reasonable to take a syntax-directed approach [Rub83,Wood86]. That is, that the structure of the specification grammar as opposed to some separate format specification, as well as the explicit formatting directives control the appearance of the output. This added fuel to the decision to use the specification parse tree to drive the prettyprinter.

The observant reader will probably have noted the slightly dubious tone of the previous paragraphs, and might wonder if the decision to mix formatting directives in with the grammar is somehow in doubt. Future implementations might want to consider separating structure specification and formatting specification. Unfortunately there is no immediate answer to this question but only more questions. First of all, current experience shows no difficulties with this decision. That means that for the work that has been done with GRAFS, it was possible (and in fact, easy) to get the formatting effects desired. However, we could conceive of a situation where the user might want to use different formatting conventions on the same grammar. Given that grammar specification and file type are rather closely tied together, this might prove difficult.

The final source of inspiration for the GRAFS prettyprinter came from the Pascal MPS prettyprinter [Cam87b]. It is the notion of *parameterization*, and is dealt with in more detail by Cameron [Cam87a]. In essence, parameterizing the prettyprinter means abstracting the prettyprinter algorithm, as much as possible, from various machine- and situation-dependent considerations. This is done by having the user provide the low-level printing action routines. These are "PrettySpaces", "PrettyNewLine", "PrettyString", "EntryMonitor", and "ExitMonitor". PrettySpaces is called by the prettyprinter to print some k number of spaces (blanks). PrettyNewLine is called when the prettyprinter wants to cause a line break. PrettyString is called to print out some text string. EntryMonitor is called each time the prettyprinter enters a new node of the parse tree being printed and ExitMonitor is called each time the prettyprinter leaves a node.

There are two reasons motivating prettyprinter parameterization. The first is that the prettyprinter may be applied in several different environments. Interactive structure editors require prettyprinted output to a CRT, whereas other applications may require prettyprinted output to a file. Parameterization allows the user to supply routines that support the application that is required. The second reason is that parameterization allows the user to have more control of the prettyprinting process. Some awkward problems can be solved by constructing customized formatting routines. For example, consider the situation where the user wishes to alter the textual representation of tokens. A particular example of this occurs in Pascal, where some compilers use a ' ^ ' to represent pointer references, and some use a '@'. A particular grammar will use only one symbol. Hence, in order to create prettyprinters that will produce output that is acceptable to either compiler, the programmer will have to alter the representation of pointer references. Parameterization makes this possible.

4.2.7 User Interface

The GRAFS user interface was designed to satisfy two basic criteria. First of all, the interface had to contribute to the readability of the application program. This was satisfied by using descriptive names for fixed GRAFS routines, and names that reflected the grammar structure that was being manipulated, in the case of generated GRAFS routines. Second, the interface had to be as simple as possible. This was satisfied by making GRAFS do as much as possible, and having it keep track of as much information as possible.

4.2.8 Type Checking

Type checking in GRAFS has two levels. The first ensures that we are dealing with the correct grammar and the second ensures that the operation to be performed on a given structure is appropriate for that structure. All type information is set, checked, and hidden by GRAFS, so there is no opportunity for the programmer, through error or intent, to subvert the type checking mechanisms.

The two levels of type checking have separate mechanisms. The grammar type of a node is stored in each node as it is created by either the Make routines or the generic node constructors that are used by the parser and the restore utility. Any time a node is to be manipulated, the manipulating routine checks the node's grammar type to confirm that that grammar type is to the one that the routine was generated from. This has two effects. It confirms that the requested operation is appropriate to the grammar and it guarantees that nodes of different grammar types cannot be mixed together into the same structure. The grammar type checking mechanism is quite simple. Each node contains the grammar type. This information is compared with the grammar type that the access routines are expecting.

The node code is, like the grammar type, also stored in each node record and is inserted by Make routines or by the parser or restore utilities. The difference is that while grammar names must be unique, node codes are only unique to a particular grammar. That is, grammars G1 and

G2 will both have a node code of 1. Thus, grammar type and node codes must both be checked.

Node codes are generated internally by the GRAFS symbol table utility. After the *gen* program parses a GRAFS specification and constructs the parse tree, the symbol table utility scans through the parse tree and assigns codes to each different rule. A rule defines the structure and hence the type of a particular node. Hence any node created according to that rule is assigned the code for that rule. This code is then checked by the manipulating routines to ensure that the requested operation corresponds to the structure to be manipulated by that routine. Consider the effect of selecting a particular field, from a node that has no such field. If the programmer is unlucky, the selection will take place resulting in a bug that could be difficult to find.

4.2.9 Structured File Input and Output

Structured file input/output is handled by the Checkpoint and Restore utilities. Checkpoint takes a GRAFS data structure and stores it, in structured form, in a file. Restore reconstructs a GRAFS data structure from a structured file. The only information that the application program needs to supply to these routines is the root node of the structure and the name of the file. In order to save a parse tree, GRAFS has to know its root, and in order to restore a parse tree, GRAFS has to know where to put the root.

Simple structured I/O is not that big a problem to implement. The prototype GRAFS uses a single data structure to construct parse trees; the *node*. Thus basic I/O is handled by constructing routines to store and recover node data structures. It should be noted that we have not explored methods of minimizing the storage of node information. Given that the information stored in a node may change, since it is not clear what information has to be stored in a node, we chose not to worry to much about storage compaction for the time being.

The more interesting part of structured I/O is the preservation of the relationships between nodes of the parse tree. The nodes have a relationship to each other. In the first stages of the GRAFS implementation, we were going to handle those relationships with pointers. However, there is one major problem with the use of pointers; it ties you to a particular address space and a particular program execution. A pointer is only valid for one computer and only for the program execution during which the pointer was assigned. It may be that case that a pointer is the same on two different machines or program executions, but will not be so in general. Given that the structures that GRAFS is to manipulate are intended to persist between program invocations and might be transferred between or manipulated by, different machines, we wanted to remove the dependence on pointers.

The solution chosen was to add a level of indirection. We created a virtual address space for each grammar. Each node that was created for a particular grammar was given a unique (to that grammar) number. That number is then used, in concert with the grammar type, to identify the node. The grammar-support subsystem of GRAFS keeps track of all grammar node numbers and handles requests for a particular node.

This solution has a number of pleasant effects. No pointer translation is required. That is, saving a particular node and then restoring that node does not require changing any of the nodes that refer to it. This saves quite a bit on implementation time and system complexity.

A parse tree does not have to be on a given machine all at once or, in fact, at all. Since the node relationships are independent of the machine address spaces or storage device address space, GRAFS can keep track of the location of nodes and move them around without having to worry about appraising referencing nodes or application programs of the actual location of a node. For example, this means that GRAFS can eventually handle data structures that are too large to be completely contained in machine memory.

Unfortunately, this level of indirection does introduce the overhead of having to translate node numbers to real addresses. However, this overhead can be minimized by using hashing techniques. Another thing that could be done is to do the address translation only on I/O. That is, when a node is read in, its virtual address is converted into a real one and when the node is written out its real address is converted back to a virtual one. This would have the advantage of lessening the address translation overhead. Given the research orientation of the prototype GRAFS, we felt that the tradeoff of performance for flexibility was worthwhile. In fact, to maintain the flexibility of GRAFS-style systems we conjecture that some sort of virtual scheme will be required. Whether or not the method that we have used will prove to be optimal remains to be seen.

4.2.10 The Complete Address Book Example Grammar

The complete grammar for our address book example is given in Figure 17. As we noted previously, the address book is just a list of records. Each record contains a name, address, phone number and occupation. The first three are fairly straightforward. The only interesting thing of note is the use of the NTS and TS directives in the `Phone` construct. These directives turn off the token spacing so a phone number is printed as "937-1445", for example, instead of as "937 - 1445". The first three fields tend to be unique to a record. That is, it is unlikely that two people in the book have the same name or the same address. Therefore, we have added the occupation field. This information allows you to write an application that could print out views of the addressbook. For example, suppose that you wanted a list of phone numbers of faculty members. The application code might look something like this:

```
addressBook := RecoverParseTree ( "AddressBook" ) ;
FOR i := 1 TO ListLength ( addressBook ) DO
 IF ProfessorQ (
  OccupationOf ( NthElement ( i, addressBook ) ) ) THEN
  PrettyPrint ( NameOf ( NthElement ( i, addressBook ) ) ) ;
  PrettyPrint ( PhoneOf ( NthElement ( i, addressBook ) ) ) ;
 END ;
END ;
```

All that this code fragment does is scan through the entries in the address book (it first reads in the structured-file "AddressBook") and if the entry is for a professor, prettyprints the name and the phone number. This fragment uses the default prettyprinter for simplicity. The result will be a column of interleaved names and phone numbers, each on its own line. If a more readable list is required, the parameterized prettyprinter `Unparse` can be used instead.

```
GRAMMAR

LIST <Records> OF <Record> SEPARATOR " " LB LB

CONSTRUCT <Record> IS "Name:" <Names:Names> LB
  "Address:" <Address:Address> LB
  "Phone:" <Phone:Phone>  LB
  "Occupation:" <Occupation:Occupation>

LIST <Names> OF <Name> SEPARATOR " "

ALTERNATE <Name> IS <Identifier> | <Initial>

LEXEME <Identifier> IS #Letter# { #Letter# }

LEXEME <Initial> IS #Letter# "."

CONSTRUCT <Address> IS ID [ "APT" <Apartment:Number> ]
  <StreetNumber:Number> <Street:Identifier> LB
  <Town:Identifier> <Country:Identifier> <Code:Code> OD

LEXEME <Number> IS #Digit# { #Digit# }

LEXEME <Code> IS (#Letter# | #Digit#) {#Letter# | #Digit#}

CONSTRUCT <Phone> IS [ "(" <Area:Number> ")" ]
  NTS <Prefix:Number> "-" <Suffix:Number> TS

ALTERNATE <Occupation> IS <Student> | <Professor> | <Other>

CONSTRUCT <Student> IS "STUDENT"

CONSTRUCT <Professor> IS "PROFESSOR"

CONSTRUCT <Other> IS "OTHER"

LEXEME #Letter# IS #UppercaseLetter# | #LowercaseLetter#

LEXICAL-CLASS #UppercaseLetter# IS "A" "B" "C" "D"
  "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P"
  "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"

LEXICAL-CLASS #LowercaseLetter# IS "a" "b" "c" "d"
  "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p"
  "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"

LEXICAL-CLASS #Digit# IS "0" "1" "2" "3" "4" "5" "6" "7"
  "8" "9"
```

Figure 17: The Complete Address Book Grammar.

One advantage to using a grammar for this example, is the ability that GRAFS has to check the input. For example, if you were entering information into the address book, GRAFS will only accept input that is syntactically correct. In other words, GRAFS will not be able to know if the phone number that you have entered is a correct one, but it will be able to check that that phone number at least appears to be correct. If you try entering "ABC-4566", GRAFS will reject it since phone number prefixes can only contain numbers. It is possible to specify things a bit more tightly, and design the grammar such that it would only accept phone number prefixes with exactly three digits. However, then you need to know that every phone number will be structured that way. Thus there is some tradeoff in design, between flexibility and the systems ability to detect anomalous input.

CHAPTER 5

GRAFS PROTOTYPE EVALUATION

This chapter looks at an example of using GRAFS. The first section considers the objectives of this research, some motivations and criteria for selecting an example, and describes the chosen example. The second section contains some observations made while the example was being implemented. The idea is to look at what GRAFS is like to work with and what some of the benefits and problems of this approach are. The third section contains an evaluation of the example application. Is the application any good and what effects did GRAFS have on its design and implementation? The last section contains a summary of the results of developing an application program using GRAFS. It deals with the details of the application, evaluates that application, and discusses the effect on the application, of using GRAFS as an implementation tool. This last section also considers these effects as to whether they are artifacts of the GRAFS implementation or model.

5.1 Prototyping a Lexical Analyzer Generator

One of the objectives of this research was to evaluate the applicability and functionality of GRAFS. One way to perform this evaluation is to use GRAFS in an application. Further, it would be useful if the application was implemented both with and without GRAFS, in order to get some feel for how effective GRAFS is. This comparison results in a subjective evaluation since only one person is involved and knowledge gained from the first implementation can be applied to the second. Also, there is more familiarity with conventional implementation than with GRAFS implementation.

With the above considerations in mind, we have chosen to re-implement a lexical analyzer generator. Just such a device was implemented for (and is a part of) GRAFS. Now the problem is to construct this device using GRAFS and make whatever comparisons, observations and conclusions that are appropriate.

The original GRAFS lexical analysis sub-system has two parts. The first part is a generator routine that takes a set of regular expressions and constructs a tabular representation of the corresponding deterministic finite automaton. The second is a table interpreter that reads the tables produced by the generator and accepts or rejects input strings accordingly. The new implementation also uses a table interpreter approach, follows the strategies and algorithms of the original code as closely as possible, and, in fact, re-uses as much of the original code as practicable. However, the second implementation makes use of GRAFS grammars to specify the major data structures. Regular expression definitions are specified with one grammar (REGrammar). Finite automata, both deterministic and non-deterministic are described by the other grammar (FAGrammar). In the original implementation, deterministic and non-deterministic automata have separate and distinctly different representations. The second implementation had its data structure support provided by GRAFS whereas the first implementation had its support routines hand-coded.

The new implementation uses two grammars, one for regular expression definition and one for finite automata. The generator proceeds in several stages. The first stage accepts regular expression definitions and generates non-deterministic finite automata. The output of this stage could be written out using the finite automaton grammar (FAGrammar), but is normally passed directly to the second stage. This second stage converts a non-deterministic finite automaton into a deterministic one. Finally, the deterministic automaton is stored in a structured file. The analyzer component reads in that automaton and uses it to analyze the input.

5.2 Observations

In this section, we discuss what it is like to work with GRAFS. It is interesting to take a support tool like GRAFS from conception, through design and implementation and finally to the stage of actually trying to do something useful with that tool. During most of the time that has been spent with GRAFS, the system has been under development. From the user's viewpoint, this means that we had to work with an incomplete and quirky system; a situation that does not engender a lot of confidence. Thus, we were delighted to discover that GRAFS is not at all difficult to work with, and we are coming to have some confidence in it.

We found it relatively easy to describe data structures using the GRAFS grammar notation. The strong reflection of the semantics by the notation made it very easy to visualize the structures that would be created. Running the specification through the generator gave immediate feedback on problems of recognition, incompleteness, spelling mistakes and the like. Incomplete specification of a grammar, whether through omission or spelling mistakes is caught by GRAFS and brought to the programmer's attention. Problems with the parsability of the grammar are pointed out by GRAFS.

The central problem of design is not made easier by GRAFS. GRAFS will help you construct a system once you know what the system is to do and how the system is to do it, but is not much help prior to that. GRAFS is a support system that deals with a lot of the drudgery that takes time away from the design process. Consider that the designers of a system are interested in solving some problem or set of problems. Unfortunately, in order to solve the problem, a number of peripheral issues must be dealt with. Various support routines that have little to do with the central issues must be constructed and debugged. Several of these routines have no function except during the development of the system and may be specific to some aspect of the design. If the design changes then support routines may have to be modified. The process of evolving a design is frustrated by having to deal with peripheral issues that have little to do with the actual design. In fact, there may even be considerable resistance to design changes caused by the amount of peripheral work involved. It is not the intention to suggest that GRAFS immediately cleans up a cluttered design process. The lexical analyzer generator is not a large program and we do not assume that these techniques are applicable to any size of system. However, these techniques to seem do be effective for programming-in-the-small.

Let us look at some of the benefits that GRAFS has on the implementation process. First of all, it saves considerable time in the construction of support routines, if the programmer is using an ADT methodology. GRAFS generates these routines automatically and correctly. Second, GRAFS provides considerable support in debugging those parts of the program that have to be constructed by the programmer.

Debugging support comes in various forms. For example, it is very difficult to use GRAFS routines incorrectly and get away with it. A considerable amount of internal type checking is performed. This checking is difficult to circumvent. Unfortunately, implementing GRAFS as a subprogramming system results in a dependence on the security of the host language, in this case, Modula-2. Since Modula-2 allows its type checking to be circumvented [Wir85], GRAFS type checking can also be circumvented. However, the use of Modula-2 opaque types hides structural information. Thus, although a language implementation may allow type circumvention via type coercion mechanisms or variant record overlay techniques, the actual structure is kept hidden. This is in contrast to Pascal, where type structure is always visible.

The prettyprinter can be invoked upon any node at any time, even from the debugger (dbx). This saves the programmer the trouble of writing special data-structure printer routines that are only used a few times and are just something else that needs to be designed, implemented, debugged and then modified as things change. The prettyprinter provides a very readable output and costs very little in terms of programmer effort. All that the programmer needs to do is insert formatting directives in the original specification.

The ability to store structured information in a file can also speed debugging. Consider the case where a program runs in stages, each stage constructing a new structure from a previous one, or modifying the previous one. As implementation and testing proceeds from stage to stage, the program has to execute for longer and longer before it gets to the stage being tested. If the structural information can be stored and recovered as required, then it is only necessary to run the debugged portion of the program and store the resulting structures once. The program section under development need only restore those structures and run. The time required to construct those structures from scratch on each run is saved. This save/restore process can, of course, be done by hand but is time consuming; GRAFS provides this facility painlessly.

A last positive observation, is on the way that GRAFS affects program correctness. GRAFS generates a parser and lexical analyzer for each grammar. Any errors in syntax will be caught. The programmer does not need to be concerned about detecting anomalous input. Recall the previous comments about the primary interest of the programmer being the central problem and not the peripheral issues. Such a peripheral issue might be the user interface. Thus, the user interface might be cobbled together on the fly, in order to get things running, and never be properly completed. There is nothing more permanent than some temporary solutions! Also, the access routines only allow the construction of syntactically correct structures. The application program is thus not capable of generating a syntactically incorrect structure.

There are three problems with the current GRAFS implementation. GRAFS is incomplete, slow, and uses too much storage. These issues are not seen as permanent and are be discussed elsewhere.

5.3 Evaluation

This section is concerned with the quality of the example application and the ways in which it was influenced by GRAFS.

The new implementation of the lexical analyzer and generator is a table interpreter and generator. However, the data structure involved is not a simple state/character transition matrix as was the case in the original. The reasons for this change were the storage limitations imposed by GRAFS. The current GRAFS implementation uses a lot of storage and the storage of individual nodes is not compact. Also there is a problem with the construction of sparse data structures. There is currently no way to create a matrix without having it completely filled. A problem with table-driven lexical analyzers is that the tables become quite large, even for fairly small finite automata. Thus, in order to accommodate these problems, the data structure was modified. The result is that access to information is not quite as fast as would be the case with a matrix.

The new implementation is also slower than the original. There are three reasons for this. First, as mentioned before, the data structure is a bit slower to retrieve information from. Second, the current implementation of GRAFS is not as fast as it might be. There are several reasons for this lack of performance. There is a considerable overhead due to type checking. Specifically, the grammar checking mechanism is not well implemented. This is not seen as a problem, since other techniques are available to speed this up considerably. The last reason, is that GRAFS currently supports only one base data type; character strings. Thus, numeric data must be converted to and from string representation every time it is to be stored or manipulated. Again, this is not a permanent situation, since we see no difficulty in extending GRAFS to support other base types, thereby removing this conversion cost.

One limitation of that will affect application programs is the way that GRAFS uses storage. Consider the simple record shown in Figure 18. Integers are stored in 4 bytes. Records fields are stored more or less contiguously, so a variable of type `Simple` should use 16 bytes. Compare that with the equivalent GRAFS construct shown in Figure 19 and diagrammed in Figure 20. Thus GRAFS adds considerable storage overhead. At the moment, GRAFS uses a homogeneous storage structure, the `node`. There is no particular reason while nodes have to be all the same. Nodes could be custom built to only hold the information that is necessary for the structure that is being represented, thus saving some storage. Also, lexical nodes could be compressed into their parent nodes to save more storage. These changes will have to wait until a future implementation.

```
Simple = RECORD
  int1 : INTEGER;
  int2 : INTEGER;
  int3 : INTEGER;
  int4 : INTEGER
  END ;
```

Figure 18: A Simple Modula-2 Record.

Now that we have mentioned the bad effects of GRAFS and made some attempt at excusing them, let us look at the positive side. It was very easy to convert and debug the original implementation to get the new one. This was much faster than constructing the original even considering the increased understanding of the problems and the reuse of code. The code required considerable modification, only the main structure remained. Bugs that took some time to find in the original came out somewhat faster with the aid of GRAFS. This was especially due to the availability of the prettyprinter for studies of incomplete structures. For example, the program would occasionally get lost prior to completing the tables. The program could be halted, and the prettyprinter called on the partially finished structure to get an idea of what was going on.

The experience of implementing the lexical analyzer twice, both times using a table driven approach, has caused us to question the desirability of this approach. It is our intention to try a re-implementation, at some point, using code-generation techniques, as was done in [Moss86]. In

```
CONSTRUCT <Simple> IS
  <int1:Integer>
  <int2:Integer>
  <int3:Integer>
  <int4:Integer>

LEXEME <Integer> IS #digit# { #digit# }
```

Figure 19: GRAFS Construct of the Figure 18 Record.

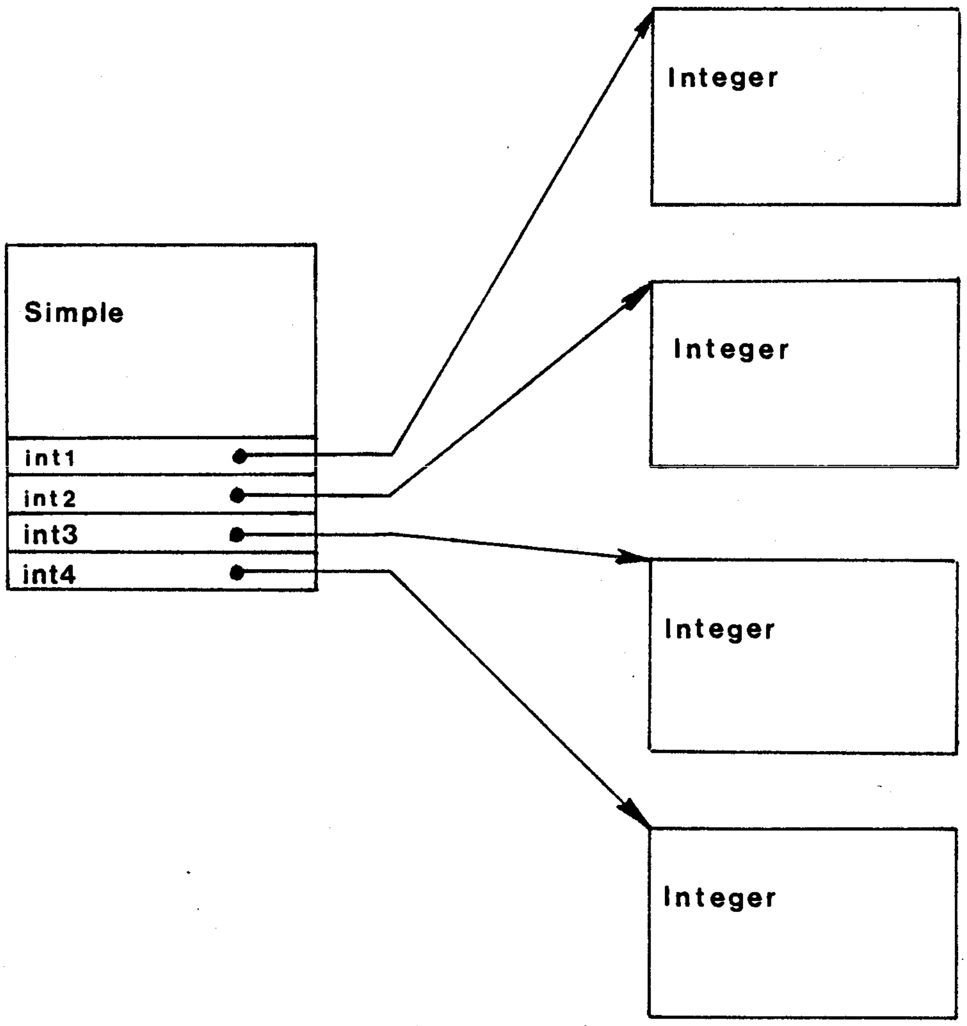

Figure 20: GRAFS Storage for Figure 19.

other words, instead of generating tables for a universal table interpreter, we would like to try generating a customized lexical analyzer routine directly from the specification. Interestingly, if the GRAFS parser generator and parser are enhanced enough to allow the GRAFS grammar to describe Modula-2, then GRAFS could generate a Modula-2 metaprogramming system. Such a metaprogramming system would support the code generation approach that we wish to use.

The new implementation was also an exercise in learning to use GRAFS. Even though we had built GRAFS and understood how it worked, we still had to learn how to design structures and specify them with a grammar. It turned out that grammar design problems were relatively easy to recover from. That is, incorrect structural or interface choices were not hard to correct. Changes were simply a matter of changing the grammar, running the grammar through the generator, and modifying the application program. Naturally, the severity of the modifications to the application program depended upon the severity of the structural modifications. Most of the time, the structure modifications were minor and the program modifications were correspondingly minor.

Table 1: Comparison of Lines of Code in the Lexical Analyzers.

Routine Class	GRAFS Internal Analyzer	Example Analyzer (using GRAFS)
RE -> NFA	308	281
NFA -> DFA	150	244
Support	304	103
Prettyprinting	188	0 (GRAFS)
Chkpt & Restore	51 (not shown)	0 (GRAFS)
Interpreter	231	310
Total	1232	938

One of the expected benefits of using GRAFS is a reduction in the amount of code that is constructed manually by the programmer. This benefit manifested itself in the example program. Table 1 shows the number of lines of code in the original (internal to GRAFS) analyzer implementation and in the new (making use of GRAFS) implementation. The table breaks the two implementations down into their components and shows the number of lines of code in each stage. It should be noted that, although some support routines needed to be constructed in the new implementation, the amount was reduced considerably. Also, prettyprinting and data structure checkpoint/restore operations were completely provided by GRAFS. The result is that the lexical analysis programs constructed using GRAFS required approximately 24 percent fewer lines of Modula-2 code than the lexical analysis sub-system of GRAFS. That 20 percent consisted of data structure support routines that were provided by GRAFS. The conclusions that can be drawn from this are limited by the inadequacy of lines-of-code as a software metric. For example, we have not rigorously dealt with problems of the equivalence of the two implementations, and we are not guaranteed that they have the same quality of code (assuming that we could agree on what that means). However, given that the two implementations were done by the same person, it seems reasonable to assume that there is some basis for comparison.

Another problem with evaluations, is that GRAFS applications can be difficult to compare with non-GRAFS applications. Consider the protection that the type checking mechanism, or that the parser provides complete syntax checking of input. It is unlikely that non-GRAFS applications would be equivalent in this area. Also, consider the level of abstraction used in the application. Programs with a lot of subroutine calls tend to run more slowly than programs with fewer subroutines calls. Hence, a programmer who does not use abstraction techniques can probably produce a faster program than someone who uses those techniques. However, problems may arise when trying to maintain a system that was implemented without abstractions.

5.4 Results

This section contains a summary of the results obtained by using GRAFS to re-implement a lexical analyzer generator.

The current implementation of GRAFS is incomplete, slow, and uses too much storage. Thus, GRAFS is not yet at the stage of being usable for production systems. However, there is no evidence to suggest that the implementation has to remain incomplete, always be slow, or that the current storage requirements will persist. For example, the current grammar-type checking mechanism is clumsy and can be sped up considerably.

Many other avenues also remain to be explored. Work done by Heuring [He86], and MossenBock [Moss86] suggests that the lexical analyzer can be made faster. Using a generated code approach instead of a table-interpreter would remove the overhead of reading in and initializing lexical analyzer data structures. Work remains to be done in the area of storage compaction. For example, Cameron [Cam86] has used syntactic information to encode program source code and store it more compactly. Hashing techniques can be used in the internal GRAFS tables. The list goes on. Also, recent work by Lamb [Lamb87] on a system that has some similarities to GRAFS indicates that such systems can approach production requirements.

We felt that using GRAFS sped program development. There were several ways in which GRAFS aided in debugging and, in addition, it saved the programmer the effort of constructing the required support routines. Also, GRAFS does enforce data abstraction. We have noted that, in cases where the programmer is responsible for all the implementation details, abstraction may not be used.

CHAPTER 6

CONCLUSIONS AND FURTHER RESEARCH

This chapter summarizes the results of this research. The first part is a discussion of what was learned about GRAFS and the second part considers the work that remains to be done.

6.1 Conclusions

The aim of this research was to consider GRAFS in terms of feasibility, applicability and productivity. Feasibility is the question of "Can we build one?". Applicability means "What can we do with it?". Productivity is a question of "Is it worth using?" or What do we gain by using it?". This section addresses these questions in the light of the research that has been done.

The question of feasibility can be disposed of fairly quickly. The answer is that it is possible to construct GRAFS. The evidence for this is that a functioning prototype was, in fact, constructed.

Chapter 3 discussed the ways in which GRAFS could be used to describe arbitrary data structures. The existing implementation covers a subset of the data structures available in common programming languages. The range of base types is currently restricted to character strings, but we strongly conjecture that that range can be extended to cover the commonly available base types. Simulation (or the effect) of pointers can be achieved with labelling techniques. Thus a full data structure description capability should be achievable by this class of system. Thus, we conclude that GRAFS will have applicability wherever complex data structures are required. Whether it will be the method of choice remains to be seen.

In our experience, if a programmer is using abstraction techniques then the implementation will proceed more quickly if GRAFS is used than if not. The reasons for this are several. First, data structure support routines are provided or generated automatically, eliminating the time required to design, implement, and debug them. Second, the enhanced type checking makes it more difficult to misuse data structures. Third, the availability of the prettyprinter and the enhanced type checking speed the debugging process. Fourth, GRAFS' grammar-based orientation makes it difficult (if not impossible) to create syntactically incorrect structures due to input errors or interface misuse.

The prototype GRAFS is incomplete, slow, and uses too much storage. However, we conjecture that these are artifacts of the prototype, and need not be the case with such a system. Furthermore, in view of our own experiences with GRAFS we feel that systems of this class have the potential to be extremely useful and that further research in this area is warranted.

6.2 Further Research

The model of GRAFS given in Chapter 3 coupled with the incompleteness of the current implementation leaves room for further work. This section outlines some of the ideas for research based on issues internal to GRAFS and those that are peripheral but closely related.

We need to expand on the associated semantics of lexical nodes. At the moment, lexical node information can only be in the form of character strings. However, there is no strong reason why they have to be so limited and there appear to be some good reasons why they should not be. The question then arises of what the semantics might be and how they might be specified. That is, do we attribute base types to lexical rules or do we attempt to provide a facility for inventing base types?

Consider grammar design issues. Does the existing GRAFS grammar give sufficient expressive power? For example, there is no explicit way to construct matrices in GRAFS notation. The best that we can do is construct lists of lists, but there is nothing to say that all lists must be the same length.

Prettyprinting and formatting require further investigation. Should the grammar contain all of the information about how to format the structures that it describes? Should there be several different format descriptions and if so how would these tie to the original grammar? What formatting operators are necessary (peripheral issue) and is it possible for the programmer to define his own operators?

Several questions exist about the file system. Currently, structures exist in named entities called files. These files exist in the context of a directory or hierarchy of directories. Could this be handled by GRAFS as a series of hierarchical tree structures? How would we handle naming of specific instances of GRAFS structures? What about optimizing the GRAFS save and restore routines?

There is a major database research issue that also arises in GRAFS. How can we deal with minor grammar changes? Grammar version 1 is changed to give version 1.1. Is it necessary to explicitly convert or can the new grammar subsume the old one in some automatic fashion?

At the moment, GRAFS works in a single process on a single machine. What about multi-process multi-machine versions? GRAFS itself has nothing to do with distributed computing, however, GRAFS is a programming support tool and programmers are working in distributed environments. It therefore seems worthwhile to look at the implementation issues involved in making GRAFS work in a distributed system.

APPENDIX A: THE GRAFS USER INTERFACE

This section deals with some of the details that are necessary to actually use GRAFS as currently implemented. These details may overlap some of what has been presented in the body of the thesis, but are presented here together to form a kind of limited user manual.

The GRAFS prototype is implemented in Modula-2 and runs on the SUN UNIX 4.2 release 3.2 operating system. The GRAFS interface consists of two parts. The first is a generator program, `gen`, that takes a grammar specification and produces the grammar-specific routines and tables. The second part is the program interface that is composed of a set of Modula-2 subroutines that are used by the user application to interface with GRAFS.

1 Generation of Grammar Routines

GRAFS is directly involved in two phases of the design process: design and implementation of major data structures, and then application program implementation. The first of these phases splits into two parts, designing and describing each major data structure, and then implementing each. The second phase involves integrating the GRAFS interface routines into the application program.

Once a data structure is decided upon, it must be described in a form acceptable to GRAFS. This is done with the notation described by the GRAFS metagrammar in the last section of this appendix. The user writes that description into a file, called, for example, "Sample". That file is then handed to the `gen` program as follows:

```
%gen -v Sample
```

The `-v` switch is optional, and will, if used, cause `gen` to produce a human readable symbol table, production table and parser table and direct them to *stdout*. These should not usually be needed, but they are available for debugging or the curious. Regardless of the setting of `-v`, `gen` will produce the following grammar specific files:

`Sample_GS.def`	-Modula-2 definition module.
`Sample_GS.mod`	-Modula-2 implementation module.
`Sample_ParseTree`	-grammar parse tree.
`Sample_NodeData`	-other grammar data.
`Sample_Tables`	-parser and lexical analyzer tables.
`Sample.list`	-input listing file plus syntax errors, parser problems etc.

(Note: GS is an abbreviation for Grammar-Specific.)

The `.list` file contains a listing of the input grammar and any errors found. If there is a syntax problem, GRAFS will tell the user what it was expecting and approximately where the problem occured. Any problems with parser table generation will also be included at the end of this file. Thus, after running `gen`, the user should check the `.list` file for any problems before proceeding any further.

The grammar-specific subroutine definitions and implementation module are files `<grammarName>_GS.mod` and `<grammarName>_GS.def` . These will need to be compiled and linked to the user application.

The remaining three files are not of direct interest to the user. They contain information that GRAFS will read as required to perform any grammar-specific operations such as parsing and prettyprinting. The user need only be concerned with these files being in the directory where the application is being used. GRAFS will find and load them as required without further effort by the user.

Using the GRAFS interface is quite simple. The programmer need only include the following modules:

```
(* data structures needed to talk to GRAFS *)
FROM DataTypes IMPORT Node, StringType ;

(* General interface routines - include what you need *)
FROM General IMPORT ParseInputFile, Prettyprint ... ;

(* Grammar-specific routines - include each routine by
(*   name or import the entire module and qualify each
(*   routine name.
FROM <GrammarName> IMPORT ... ;
```

GRAFS structures are parse trees and are composed of `Node` objects. The programmer does not have to be aware of what is in a node, but he will have to allocate storage for them, pass them to GRAFS, and receive them from GRAFS. `StringType` is the GRAFS type for handling character strings. The programmer allocates storage of that type in the same ways as for `Node`. The only difference is that `StringType` is really an array and therefore can not be returned by a function, it has to be passed around as a VAR parameter.

The general interface routines that are currently available are as follows:

```
DEFINITION MODULE  General;

FROM DataTypes IMPORT Node, StringType ;

FROM PrettyPrinter IMPORT

   (* procedure - takes a single StringType argument *)
   PrettyStringProc,

   (* procedure - takes a single CARDINAL argument *)
   PrettySpacesProc,

   (* procedure - no arguments *)
   PrettyNewLineProc,
  (* procedure - takes a single Node argument *)
  MonitorProc ;

(* Parser  (*     fileName - name of file to be parsed.
(*     grammarName - name of grammar to use for parse.
(*     startSymbol - name of grammar-rule to be used as parser
(*           start symbol.
(*     result - TRUE if parse succeeded, FALSE otherwise.
(*   RETURN VALUE : root Node of constructed parse tree.
(*
PROCEDURE ParseInputFile (
  fileName, grammarName, startSymbol : StringType ;
  VAR result : BOOLEAN
  ) : Node ;

(* UnParsers *)
(* PrettyPrint - quick and dirty pretty printer. Uses default
(*              formatting routines and prints to stdout.
(*
PROCEDURE PrettyPrint ( n : Node ) ;

(* UnParse - like PrettyPrint but you get to roll your own
(*          formatting routines.
(*
PROCEDURE UnParse ( n : Node ;
    (* string printer *)
    PrettyString : PrettyStringProc ;
    (* number of spaces printer *)
    PrettySpaces : PrettySpacesProc ;
    (* new line printer *)
    PrettyNewLine : PrettyNewLineProc ;
    (* EntryMonitor is called as the prettyprinter
    (*        enters each node,
    (* ExitMonitor is called as the prettyprinter
    (*        leaves each node.
    (*
    EntryMonitor, ExitMonitor : MonitorProc ) ;

(* List manipulation *)
(* NthElement - retrieves the ith child of the list x1.
(*
PROCEDURE NthElement ( i : CARDINAL ; x1 : Node ) : Node ;
(* ListLength - returns the number of children
(*          in the list x.
(*
PROCEDURE ListLength ( x : Node ) : CARDINAL ;
(* AppendNodeToList - adds node to the end of list. *)
PROCEDURE AppendNodeToList ( list, node : Node ) ;

(* Generic Predicates *)
(* EmptyNodeQ - returns TRUE if the node exists,
(*          FALSE otherwise.
(*
PROCEDURE EmptyNodeQ ( x : Node ) : BOOLEAN ;

(* Post Processing call *)
(* PostProcessing - clean up at the end, application programs
(*                should call this routine just before shutting
(*                down.
(*
PROCEDURE PostProcessing () ;

(* Save/Recover Parse Tree Interface *)
(* SaveParseTree - saves the parse tree at root n, in the
(*               file, fileName.
(*
PROCEDURE SaveParseTree ( n : Node ; fileName : StringType ) ;
(* RecoverParseTree - reads in the file, fileName, reconstructs
(*            the parse tree contained therein and returns
(*            the root node to the calling routine.
(*
PROCEDURE RecoverParseTree ( fileName : StringType ) : Node ;

END General.
```

3 The GRAFS Metagrammar

This section contains a description of the GRAFS metagrammar in terms of itself. Thus, the following grammar defines the grammars that are acceptable syntactically, to GRAFS.

```
GRAMMAR

  CONSTRUCT <Grammar> IS "GRAMMAR" LB <Rules:RuleList>

  LIST <RuleList> OF <Rule> SEPARATOR " " LB LB

  ALTERNATE <Rule> IS <ConstructionRule> |
      <AlternationRule> | <RepetitionRule> |
      <LexicalRule> | <LexicalClassRule>

  CONSTRUCT <ConstructionRule> IS ID "CONSTRUCT"
      <Name:SimpleNonTerminal> "IS"
      <ConstructionElements:ConstructionElementList>
      OD

  CONSTRUCT <AlternationRule> IS ID "ALTERNATE"
      <Name:SimpleNonTerminal> "IS"
      <Alternatives:SimpleNonTerminalList> OD
```

```
CONSTRUCT <RepetitionRule> IS ID "LIST"
    <Name:SimpleNonTerminal>
    "OF" <BaseType:SimpleNonTerminal> "SEPARATOR"
    <Separator:Terminal> <Formattors:DirectiveList>
    OD

CONSTRUCT <LexicalRule> IS ID "LEXEME"
    <Name:GenericNonTerminal>
    "IS" <Definition:RegularExpression>
    [ <Delimiter:DelimiterExpression> ] OD

CONSTRUCT <DelimiterExpression> IS "L-DELIMITER"
    <LDelimiter:OneCharacter>
    "R-DELIMITER" <RDelimiter:OneCharacter> OD

CONSTRUCT <LexicalClassRule> IS ID "LEXICAL-CLASS"
    <Name:LexicalNonTerminal> "IS"
    <Members:CharacterList> OD

LIST <ConstructionElementList> OF
    <ConstructionElement> SEPARATOR " "

ALTERNATE <ConstructionElement> IS <Terminal> |
    <CompoundNonTerminal> | <OptionalPhrase> |
    <Directive>

CONSTRUCT <OptionalPhrase> IS ID "["
    <Head:BasicElementList>
    <Body:CompoundNonTerminal>
    <Tail:BasicElementList> "]" OD

LIST <BasicElementList> OF <BasicElement>
    SEPARATOR " "

ALTERNATE <BasicElement> IS <Terminal> |
    <Directive>

LIST <SimpleNonTerminalList> OF
    <SimpleNonTerminal> SEPARATOR "|"

LIST <CharacterList> OF <OneCharacter>
    SEPARATOR " "

ALTERNATE <RegularExpression> IS <Term> |
    <AlternationExpression>

ALTERNATE <Term> IS <Factor> |
    <ConcatentationExpression>

ALTERNATE <Factor> IS <Terminal> |
    <ClosureExpression> | <LexicalNonTerminal>
    | <BrackettedExpression>
```

```
CONSTRUCT <AlternationExpression> IS ID
    <Operand1:Term> "|" <Operand2:RegularExpression>
    OD

CONSTRUCT <ConcatenationExpression> IS ID
    <Operand1:Term> <Operand2:Factor> OD

CONSTRUCT <ClosureExpression> IS ID "{"
    <Operand:RegularExpression> "}" OD

CONSTRUCT <BrackettedExpression> IS ID "("
    <Operand:RegularExpression> ")" OD

CONSTRUCT <CompoundNonTerminal> IS NTS "<"
    <Component:Identifier> ":" <Class:Identifier>
    ">" TS

ALTERNATE <GenericNonTerminal> IS <SimpleNonTerminal>
    | <LexicalNonTerminal>

LIST <DirectiveList> OF <Directive> SEPARATOR " "

ALTERNATE <Directive> IS <IDRule> | <ODRule> |
    <LBRule> | <TSRule> | <NTSRule>

CONSTRUCT <IDRule> IS "ID"

CONSTRUCT <ODRule> IS "OD"

CONSTRUCT <LBRule> IS "LB"

CONSTRUCT <TSRule> IS "TS"

CONSTRUCT <NTSRule> IS "NTS"

CONSTRUCT <LexicalNonTerminal> IS NTS "#"
    <Identifier:Identifier> "#" TS

CONSTRUCT <SimpleNonTerminal> IS NTS "<"
    <Identifier:Identifier> ">" TS

ALTERNATE <Terminal> IS <OneCharacter> |
    <CharacterString>

LEXEME <OneCharacter> IS
    ( #Character# | #Delimiter# )
    L-DELIMITER "\""
    R-DELIMITER "\""

LEXEME <Identifier> IS #Letter# { #Letter# | #Digit# }

LEXEME <CharacterString> IS #Character# #Character#
```

```
    { #Character# }
    L-DELIMITER "\"" R-DELIMITER "\""

LEXEME #Letter# IS #UppercaseLetter# |
    #LowercaseLetter#

LEXICAL-CLASS #UppercaseLetter# IS "A" "B" "C" "D"
    "E" "F" G" "H" "I" "J" "K" "L" "M" "N" "O" "P"
    "Q" "R" "S" "T" "U" "V" W" "X" "Y" "Z"

LEXICAL-CLASS #LowercaseLetter# IS "a" "b" "c" "d"
    "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p"
    "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"

LEXICAL-CLASS #Digit# IS "0" "1" "2" "3" "4" "5"
    "6" "7" "8" "9"

LEXICAL-CLASS #Special# IS "." "," "!" "@" "#" "$"
    "%" "^" "&" "*"
    "(" ")" ";" ":" "_" "-" "|" "+" "=" "[" "]" "\\"
    "/" "{" "}"
    "<" ">" "?" "~" " " "'"

LEXICAL-CLASS #Character# IS "A" "B" "C" "D" "E" "F" "G" "H"
    "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V"
    "W" "X" "Y" "Z"
    "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
    "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
    "0" "1" "2" "3" "4" "5" "6" "7" "8" "9"
    "." "," "!" "@" "#" "$" "%" "*" "^" "&" "("
    ")" ";" ":" "_" "-" "|" "+" "=" "[" "]" "\\"
    "/" "{" "}" "<" ">" "?" "~" " " "'"

LEXICAL-CLASS #Delimiter# IS "\""
```

APPENDIX B: EXAMPLE LEXICAL ANALYZER GRAMMARS AND GRAMMAR INTERFACES

This appendix contains the grammars, grammar-specific interface routines and main source code for the GRAFS lexical analysis package implementation as well as the source code for the GRAFS lexical analysis sub-system (the one internal to GRAFS).

Sections 1 and 2 contain the regular expression grammar and its interface. Sections 3 and 4 contain the finite automata grammar and its interface. Section 5 contains the lexical generator source code for both implementations. It is organized in a side-by-side manner, sideways on the page (to get a reasonable code width), with the old implementation on the left side and the new one on the right. Section 6 contains the lexical analyzer source code for both implementations. As with Section 5, the two implementations are laid side-by-side with the old implementation on the left and the new one on the right. We have made some attempt to align corresponding sections of the code to ease comparison.

1 Regular Expression Grammar

This grammar describes a structure for regular expressions.

```
GRAMMAR

   CONSTRUCT <REInput> IS <REList:REList> <SubREList:SubREList>

   LIST <REList> OF <REDefinition> SEPARATOR " " LB LB

   LIST <SubREList> OF <SubREDefinition> SEPARATOR " " LB LB

   CONSTRUCT <REDefinition> IS "DEFINITION" <Name:Identifier>
      "IS" ID <Definition:RE> OD

   ALTERNATE <SubREDefinition> IS <SubDefinition> |
   <ClassDefinition>

CONSTRUCT <SubDefinition> IS "SUB-DEFINITION"
   <Name:Identifier> "IS" ID <Definition:RE> OD

CONSTRUCT <ClassDefinition> IS "CLASS-DEFINITION"
   <Name:Identifier> "IS" ID <Definition:CharacterList> OD

LIST <CharacterList> OF <Terminal> SEPARATOR " "

ALTERNATE <RE> IS <Term> | <AlternationExpression>

ALTERNATE <Term> IS <Factor> | <ConcatenationExpression>

ALTERNATE <Factor> IS <Terminal> | <ClosureExpression> |
  <LexicalNonTerminal> | <BrackettedExpression>

CONSTRUCT <AlternationExpression> IS <Operand1:Term> "|"
    <Operand2:RE>

CONSTRUCT <ConcatenationExpression> IS <Operand1:Term>
    <Operand2:Factor>

CONSTRUCT <ClosureExpression> IS "{" <Operand:RE> "}"

CONSTRUCT <BrackettedExpression> IS "(" <Operand:RE> ")"

CONSTRUCT <LexicalNonTerminal> IS "#" <Identifier:Identifier>
    "#"

LEXEME <Terminal> IS { #Character# }
    L-DELIMITER "\"" R-DELIMITER "\""

LEXEME <Identifier> IS #Letter# { #Letter# | #Digit# }

LEXEME #Letter# IS #UppercaseLetter# | #LowercaseLetter#

LEXICAL-CLASS #UppercaseLetter# IS "A" "B" "C" "D" "E" "F"
   "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
   "R" "S" "T" "U" "V" "W" "X" "Y" "Z"

LEXICAL-CLASS #LowercaseLetter# IS "a" "b" "c" "d" "e" "f"
   "g" "h" "i" "j" "k" "l" "m" "n" "o" "p"
   "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"

LEXICAL-CLASS #Digit# IS "0" "1" "2" "3" "4" "5" "6" "7"
   "8" "9"

LEXICAL-CLASS #Special# IS "." "," "!" "" "#" "$" "%" "^"
   ")" ";" ":" "_" "-" "|" "+" "=" "[" "]" "\\" "/" "{" "}"
   "*" "(" "&" "‾" " " "'" "?" "<" ">"

LEXICAL-CLASS #Character# IS "A" "B" "C" "D" "E" "F"
```

```
"G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R"
"S" "T" "U" "V" "W" "X" "Y" "Z"
"a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
"m" "n" "o" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
"0" "1" "2" "3" "4" "5" "6" "7" "8" "9"
"." "," "!" "" "#" "$" "%" "^" "&" "*" "("
")" ";" ":" "_" "-" "|" "+" "=" "[" "]" "\\" "/" "{"
"}" "<" "?" "¯" " " "'" "?"
```

2 Regular Expression Support Routines

The following Modula-2 definition file was automatically generated from the preceding regular expression grammar.

```
DEFINITION MODULE REGrammar_GS ;

FROM DataTypes IMPORT Node, StringType ;

CONST (* Node Codes *)
  Terminal = 1000 ;
  Identifier = 1001 ;

  REInput = 5000 ;
  REList = 5001 ;
  SubREList = 5002 ;
  REDefinition = 5003 ;
  SubREDefinition = 5004 ;
  SubDefinition = 5005 ;
  ClassDefinition = 5006 ;
  CharacterList = 5007 ;
  RE = 5008 ;
  Term = 5009 ;
  Factor = 5010 ;
  AlternationExpression = 5011 ;
  ConcatenationExpression = 5012 ;
  ClosureExpression = 5013 ;
  BrackettedExpression = 5014 ;
  LexicalNonTerminal = 5015 ;

PROCEDURE MakeREInput ( x1, x2 : Node ) : Node ;
PROCEDURE MakeREList () : Node ;
PROCEDURE MakeSubREList () : Node ;
PROCEDURE MakeREDefinition ( x1, x2 : Node ) : Node ;
PROCEDURE MakeSubDefinition ( x1, x2 : Node ) : Node ;
PROCEDURE MakeClassDefinition ( x1, x2 : Node ) : Node ;
PROCEDURE MakeCharacterList () : Node ;
PROCEDURE MakeAlternationExpression ( x1, x2 : Node ) : Node ;
PROCEDURE MakeConcatenationExpression ( x1, x2 : Node ) :
  Node ;
PROCEDURE MakeClosureExpression ( x1 : Node ) : Node ;
PROCEDURE MakeBrackettedExpression ( x1 : Node ) : Node ;
PROCEDURE MakeLexicalNonTerminal ( x1 : Node ) : Node ;
PROCEDURE MakeTerminal ( x1 : StringType ) : Node ;
PROCEDURE MakeIdentifier ( x1 : StringType ) : Node ;
PROCEDURE REInputQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE REListQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE SubREListQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE REDefinitionQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE SubREDefinitionQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE SubDefinitionQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE ClassDefinitionQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE CharacterListQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE REQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE TermQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE FactorQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE AlternationExpressionQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE ConcatenationExpressionQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE ClosureExpressionQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE BrackettedExpressionQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE LexicalNonTerminalQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE TerminalQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE IdentifierQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE REListOf ( x1 : Node ) : Node ;
PROCEDURE SubREListOf ( x1 : Node ) : Node ;
PROCEDURE NameOf ( x1 : Node ) : Node ;
PROCEDURE DefinitionOf ( x1 : Node ) : Node ;
PROCEDURE Operand1Of ( x1 : Node ) : Node ;
PROCEDURE Operand2Of ( x1 : Node ) : Node ;
PROCEDURE OperandOf ( x1 : Node ) : Node ;
PROCEDURE IdentifierOf ( x1 : Node ) : Node ;
PROCEDURE RetrieveTerminalOf ( x1 : Node ;
  VAR str : StringType ) ;
PROCEDURE RetrieveIdentifierOf ( x1 : Node ;
  VAR str : StringType ) ;

END REGrammar_GS .
```

3 Finite Automaton Grammar

This grammar describes a structure for finite automaton; both deterministic and non-deterministic.

```
GRAMMAR

  CONSTRUCT <FA> IS <StateList:StateList>
```

```
LIST <StateList> OF <State> SEPARATOR " " LB LB

CONSTRUCT <State> IS "STATE" <Number:Number> ID LB
  "TRANSITIONS" <Transitions:TransitionList> LB
  "DEFAULT" <Default:Default> LB
  "TYPE" <Type:Type> OD

LIST <TransitionList> OF <Transition> SEPARATOR " " LB

CONSTRUCT <Default> IS
  "NEXT_STATE" <NextState:Number>

CONSTRUCT <Transition> IS
  "CHARACTER" <CharacterList:CharacterList>
  "NEXT_STATE" <NextState:Number>

ALTERNATE <Type> IS <Accept> | <Reject>

CONSTRUCT <Accept> IS "ACCEPT" <Output:Number>

CONSTRUCT <Reject> IS "REJECT"

LIST <CharacterList> OF <Character> SEPARATOR " "

ALTERNATE <Character> IS <NonPrintingCharacter> |
   <PrintingCharacter>

ALTERNATE <NonPrintingCharacter> IS <epsilon> | <eoln> |
    <eof>

CONSTRUCT <epsilon> IS "EPSILON"

CONSTRUCT <eoln> IS "EOLN"

CONSTRUCT <eof> IS "EOF"

LEXEME <Number> IS #Digit# { #Digit# }

LEXEME <PrintingCharacter> IS #CharacterSet#

LEXICAL-CLASS #Digit# IS "1" "2" "3" "4" "5" "6" "7" "8"
    "9" "0"

LEXICAL-CLASS #CharacterSet# IS "A" "B" "C" "D" "E" "F" "G"
    "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U"
    "V" "W" "X" "Y" "Z"
    "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"
    "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
    "0" "1" "2" "3" "4" "5" "6" "7" "8" 9"
    "." "," "!" "" "#" "$" "%" "^" "&" ")" ";" ":"
    "_" "-" "|" "+" "=" "[" "]" "\\" "/" "{" "}"
    "?" " " " " "'" "?" "*" "(" "<" ">"
```

4 Finite Automaton Support Routines

The following Modula-2 definitions were generated from the preceding finite automaton grammar.

```
DEFINITION MODULE FAGrammar_GS ;

FROM DataTypes IMPORT Node, StringType ;

CONST (* Node Codes *)
  Number = 1000 ;
  PrintingCharacter = 1001 ;

  FA = 5000 ;
  StateList = 5001 ;
  State = 5002 ;
  TransitionList = 5003 ;
  Default = 5004 ;
  Transition = 5005 ;
  Type = 5006 ;
  Accept = 5007 ;
  Reject = 5008 ;
  CharacterList = 5009 ;
  Character = 5010 ;
  NonPrintingCharacter = 5011 ;
  epsilon = 5012 ;
  eoln = 5013 ;
  eof = 5014 ;

PROCEDURE MakeFA ( x1 : Node ) : Node ;
PROCEDURE MakeStateList () : Node ;
PROCEDURE MakeState ( x1, x2, x3, x4 : Node ) : Node ;
PROCEDURE MakeTransitionList () : Node ;
PROCEDURE MakeDefault ( x1 : Node ) : Node ;
PROCEDURE MakeTransition ( x1, x2 : Node ) : Node ;
PROCEDURE MakeAccept ( x1 : Node ) : Node ;
PROCEDURE MakeReject ( ) : Node ;
PROCEDURE MakeCharacterList () : Node ;
PROCEDURE Makeepsilon ( ) : Node ;
PROCEDURE Makeeoln ( ) : Node ;
PROCEDURE Makeeof ( ) : Node ;
PROCEDURE MakeNumber ( x1 : StringType ) : Node ;
PROCEDURE MakePrintingCharacter ( x1 : StringType ) : Node ;
PROCEDURE FAQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE StateListQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE StateQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE TransitionListQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE DefaultQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE TransitionQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE TypeQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE AcceptQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE RejectQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE CharacterListQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE CharacterQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE NonPrintingCharacterQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE epsilonQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE eolnQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE eofQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE NumberQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE PrintingCharacterQ ( x1 : Node ) : BOOLEAN ;
PROCEDURE StateListOf ( x1 : Node ) : Node ;
PROCEDURE NumberOf ( x1 : Node ) : Node ;
PROCEDURE TransitionsOf ( x1 : Node ) : Node ;
PROCEDURE DefaultOf ( x1 : Node ) : Node ;
PROCEDURE TypeOf ( x1 : Node ) : Node ;
PROCEDURE NextStateOf ( x1 : Node ) : Node ;
PROCEDURE CharacterListOf ( x1 : Node ) : Node ;
PROCEDURE OutputOf ( x1 : Node ) : Node ;
PROCEDURE RetrieveNumberOf ( x1 : Node ;
  VAR str : StringType ) ;
PROCEDURE RetrievePrintingCharacterOf ( x1 : Node ;
  VAR str : StringType ) ;

END FAGrammar_GS .
```

5 Lexical Generator Comparison

This section contains the source code for both the old and new versions of the lexical generators. The old version is on the left and the new one (using GRAFS) is on the right.

```
IMPLEMENTATION MODULE LexGen;

  FROM InOut IMPORT
    Write, WriteLn, WriteString, WriteCard, Read, Done;
  FROM Storage IMPORT ALLOCATE, DEALLOCATE;
  FROM DataTypes IMPORT StringType, Node, NodeClass;
  FROM ParseTreeModule IMPORT
    GetClass, GetComponentK, NumberOfComponentsOf, GetRule;
  FROM Generics IMPORT EmptyQ;
  FROM META_GS IMPORT
    RetrieveOneCharacterOf, RetrieveCharacterStringOf,
    RetrieveIdentifierOf, OneCharacterQ, CharacterStringQ,
    IdentifierQ, ConstructionElementsOf, NameOf, TerminalQ,
    SimpleNonTerminalQ, LexicalClassRuleQ, LexicalRuleQ,
    MembersOf, DefinitionOf, IdentifierOf, SeparatorOf,
    Operand1Of, OperandOf, Operand2Of,
    AlternationExpressionQ, ConcatenationExpressionQ,
    BrackettedExpressionQ, ClosureExpressionQ,
    LexicalNonTerminalQ, DelimiterOf, RDelimiterOf,
    LDelimiterOf;
  FROM String IMPORT Length, Compare, CompareResult;
  FROM ErrorModule IMPORT Error;
  FRDM StackModule IMPORT
    Stack, StackPush, StackPop, StackNotEmptyQ, MakeStack,
    FreeStack;
  FROM SetModule IMPORT
    Set, MakeSet, FreeSet, EnterKinSet, DeleteKfromSet,
    MemberKQ, IdenticalSetsQ, Union, ClearSet, PrintSet,
    EmptySetQ, SetState, MakeSetState, FreeSetState, First,
    Next;
  FROM GrammarSymbols IMPORT
    ERROR, FindSubAutomaton, GetFirstTokenNumber,
    GetLastTokenNumber, GetTokenString,
    GetFirstAutomatonNumber, GetLastAutomatonNumber,
    GetAutomaton, Symbols;
```

```
MODULE lexProgram;

  FROM ErrorModule IMPORT Error;
  FROM DataTypes IMPORT Node, StringType;
  FROM InOut IMPORT WriteString, WriteLn, WriteCard;
  FROM UnixParam IMPORT
    UPResult, NoOfArguments, GetArgument;
  FROM String IMPORT
    Compare, CompareResult, Length, Assign, Concat;
  FROM Storage IMPORT ALLOCATE, DEALLOCATE;
  FROM Convert IMPORT StrToCard, CardToStr;
  FROM StackModule IMPORT
    Stack, StackPush, StackPop, StackNotEmptyQ, MakeStack,
    FreeStack;
  FROM SetModule IMPORT
    Set, MakeSet, FreeSet, EnterKinSet, DeleteKfromSet,
    MemberKQ, IdenticalSetsQ, Union, ClearSet, PrintSet,
    EmptySetQ, SetState, MakeSetState, FreeSetState, First,
    Next;
  FROM General IMPORT
    ParseInputFile, PrettyPrint, UnParse, NthElement,
    ListLength, AppendNodeToList, EmptyNodeQ, SaveParseTree,
    RecoverParseTree;
  FROM FAGrammar_GS IMPORT
    MakeFA, MakeStateList, MakeState, MakeTransitionList,
    MakeTransition, MakeAccept, MakeReject, Makeepsilon,
    Makeeoln, Makeeof, MakeCharacterList, MakeNumber,
    MakePrintingCharacter, FAQ, StateListQ, StateQ,
    TransitionListQ, TransitionQ, TypeQ, AcceptQ, RejectQ,
    CharacterQ, NonPrintingCharacterQ, epsilonQ, eolnQ,
    eofQ, NumberQ, PrintingCharacterQ, StateListDf,
    NumberOf, TransitionsOf, TypeOf, CharacterListOf,
    OutputOf, NextStateOf, RetrieveNumberOf,
    RetrievePrintingCharacterOf, MakeDefault;
  FROM REGrammar_GS IMPORT
    MakeREInput, MakeREList, MakeSubREList,
    MakeREDefinition, MakeSubDefinition,
    MakeAlternationExpression, MakeClassDefinition,
    MakeConcatenationExpression, MakeClosureExpression,
CONST
  STARTCHARS = ORD(' ');
  MAXCHARS = ORD(' ') + 1;
  EPSILON = CHR(MAXCHARS);
  MAX_STATES = 220;

TYPE
  Result = (ok, Fail, error);
  Mode = (STEP, CONSTRUCT);
  CharStateMatrix =
    ARRAY [STARTCHARS..MAXCHARS], [1..MAX_STATES] OF
      CARDINAL;
  CharStateSetMatrix =
    ARRAY [STARTCHARS..MAXCHARS], [1..MAX_STATES] OF Set;
  NDATable = POINTER TO NDATableRecord;
  NDATableRecord = RECORD matrix : CharStateSetMatrix END;
  DATable = POINTER TO DATableRecord;
  DATableRecord = RECORD matrix : CharStateMatrix END;
  ActionTable = POINTER TO ActionTableRecord;
  ActionTableRecord = RECORD matrix : CharStateMatrix END;
  LexicalAnalyzerTables =
    POINTER TO LexicalAnalyzerTablesRecord;
  LexicalAnalyzerTablesRecord =
    RECORD
      stateTable : DATable;
      actionTable : ActionTable;
      startChars : CARDINAL;
      maxChars : CARDINAL;
      maxStates : CARDINAL
     END;


PROCEDURE MakeLexicalAnalyzerTables
            () : LexicalAnalyzerTables;

  VAR t : LexicalAnalyzerTables;

BEGIN
  ALLOCATE(t, SIZE(LexicalAnalyzerTablesRecord));
  t^.stateTable := MakeDATable();
  t^.actionTable := MakeActionTable();
  t^.startChars := STARTCHARS;
  t^.maxChars := MAXCHARS;
```

```
  MakeBrackettedExpression, MakeLexicalNonTerminal,
  MakeTerminal, REInputQ, REListQ, SubREListQ,
  REDefinitionQ, SubREDefinitionQ, REQ, TermQ, FactorQ,
  AlternationExpressionQ, ConcatenationExpressionQ,
  ClosureExpressionQ, BrackettedExpressionQ,
  LexicalNonTerminalQ, TerminalQ, IdentifierQ, REListOf,
  SubREListOf, NameOf, DefinitionOf, Operand1Of,
  Operand2Of, OperandOf, IdentifierOf, RetrieveTerminalOf,
  RetrieveIdentifierOf, SubDefinitionQ, ClassDefinitionQ;

CONST
  ERROR = 0;
  FAIL = 1;
  START_STATE = 2;
  MAX_STATES = 1000;
  STARTCHARS = ORD(' ');
  MAXCHARS = ORD(' ');

TYPE Result = (ok, error); Mode = (STEP, CONSTRUCT);

VAR
  argc, i : CARDINAL;
  failStr, str, grammarName, arg, name, fileName,
    errorStr : StringType;
  upResult : UPResult;
  result : Result;
  okay : BOOLEAN;
  dFA, fa, res, re, reList, subReList : Node;
  reListLen : CARDINAL;
  stateNumber, tokenNumber : CARDINAL;


PROCEDURE FindSubAutomaton
            (VAR str : StringType; subReList : Node) :
            Node;

  VAR nameStr : StringType;

BEGIN
  FOR i := 1 TO ListLength(subReList) DO
    RetrieveIdentifierOf
      (NameOf(NthElement(i, subReList)), nameStr);
    IF Compare(str, nameStr) = equal THEN
  t^.maxStates := MAX_STATES;
  RETURN (t);
END MakeLexicalAnalyzerTables;


PROCEDURE StateTableOf
            (t : LexicalAnalyzerTables) : DATable;

BEGIN
  RETURN (t^.stateTable);
END StateTableOf;


PROCEDURE ActionTableOf
            (t : LexicalAnalyzerTables) : ActionTable;

BEGIN
  RETURN (t^.actionTable);
END ActionTableOf;


PROCEDURE StartCharsOf
            (t : LexicalAnalyzerTables) : CARDINAL;

BEGIN
  RETURN (t^.startChars);
END StartCharsOf;


PROCEDURE SetStartCharsOf
            (c : CARDINAL; t : LexicalAnalyzerTables);

BEGIN
  t^.startChars := c;
END SetStartCharsOf;


PROCEDURE MaxCharsOf
            (t : LexicalAnalyzerTables) : CARDINAL;

BEGIN
  RETURN (t^.maxChars);
END MaxCharsOf;


PROCEDURE SetMaxCharsOf
            (c : CARDINAL; t : LexicalAnalyzerTables);

BEGIN
  t^.maxChars := c;
END SetMaxCharsOf;
```

```
      RETURN (NthElement(i, subReList));
     END;
   END;
  RETURN (Node(NIL));
END FindSubAutomaton;


PROCEDURE CreateNumberNode ( number : CARDINAL )
            : Node ;

  VAR str : StringType ;
      okay : BOOLEAN ;

BEGIN
  CardToStr(stateNumber, str, 0, okay);
  IF NOT okay THEN
    Error("CreateNumberNode: conversion error.");
   END;
  RETURN ( MakeNumber (str) );
END CreateNumberNode ;


PROCEDURE AddState
            (fa : Node; stateNumber : CARDINAL;
             tokenStr : StringType; mode : Mode);
BEGIN
  IF mode = CONSTRUCT THEN
    AppendNodeToList
      (StateListOf(fa),
       MakeState
         (CreateNumberNode(stateNumber),
          MakeTransitionList(),
          MakeDefault(MakeNumber(failStr)),
          MakeAccept(MakeNumber(tokenStr))));
   ELSE
    AppendNodeToList
      (StateListOf(fa),
       MakeState
         (CreateNumberNode(stateNumber),
          MakeTransitionList(),
          MakeDefault(MakeNumber(failStr)),
          MakeReject()));
   END;
END AddState;


PROCEDURE EpsilonMove
            (fa : Node;
             fromState, toState : CARDINAL);

  VAR chList : Node;
```

```
PROCEDURE MaxStatesOf
            (t : LexicalAnalyzerTables) : CARDINAL;

BEGIN
  RETURN (t^.maxStates);
END MaxStatesOf;


PROCEDURE SetMaxStatesOf
            (c : CARDINAL; t : LexicalAnalyzerTables);

BEGIN
  t^.maxStates := c;
END SetMaxStatesOf;


PROCEDURE BuildAutomaton
            (rule : Node;
             tokenNumber : CARDINAL;
             NDA : NDATable;
             NDAActions : ActionTable;
             VAR nextNewState : CARDINAL) : Result;

  VAR
    currentState, lastCurrentState, saveState, nextState :
      CARDINAL;
    result : Result;
    i, j : CARDINAL;
    delimiter, lDelimiter, rDelimiter, automaton : Node;
    mode : Mode;
    str : StringType;
    nextStateSet : Set;

BEGIN
  automaton := DefinitionOf(rule);
  mode := CONSTRUCT;
  currentState := START_STATE;
  SetNextNDAStateSet
    (NDA, EPSILON, currentState, nextNewState);
  currentState := nextNewState;
  nextNewState := nextNewState + 1;
  IF ConcatenationExpressionQ(automaton) THEN
        (* match both subexpressions *)
    result :=
      BuildAutomatonR
        (STEP,
         Operand1Of(automaton),
         tokenNumber,
         NDA,
         NDAActions,
         nextNewState,
```

```
BEGIN
  AppendNodeToList(chList, Makeepsilon());
  AppendNodeToList
    (TransitionsOf
       (NthElement(fromState, StateListOf(fa))),
     MakeTransition
       (chList, CreateNumberNode(toState)));
END EpsilonMove;


PROCEDURE BuildAutomaton
            (fa, reDefinition, subReList : Node;
             tokenNumber : CARDINAL;
             VAR nextNewState : CARDINAL) : Result;

  VAR
    currentState, lastCurrentState, saveState, nextState :
      CARDINAL;
    result : Result;
    i, j : CARDINAL;
    re, chList : Node;
    mode : Mode;
    stateStr, str, tokenStr, errorStr : StringType;
    okay : BOOLEAN;

BEGIN
  re := DefinitionOf(reDefinition);
  mode := CONSTRUCT;
  currentState := START_STATE;
  CardToStr(tokenNumber, tokenStr, 0, okay);
  IF NOT okay THEN
    Error("BuildAutomaton: conversion error.");
   END;
  AddState (fa, nextNewState, "", STEP);
  EpsilonMove (fa, START_STATE, nextNewState);
  currentState := nextNewState;
  nextNewState := nextNewState + 1;
  IF ConcatenationExpressionQ(re) THEN
        (* match both subexpressions *)
    result :=
      BuildAutomatonR
        (STEP,
         fa,
         Operand1Of(re),
         subReList,
      currentState);
  IF result = ok THEN
    result :=
      BuildAutomatonR
        (mode,
         Operand2Of(automaton),
         tokenNumber,
         NDA,
         NDAActions,
         nextNewState,
         currentState);
   END;
 ELSIF AlternationExpressionQ(automaton) THEN
      (* match either subexpression *)
  saveState := currentState;
  result :=
    BuildAutomatonR
      (mode,
       Operand1Of(automaton),
       tokenNumber,
       NDA,
       NDAActions,
       nextNewState,
       currentState);
  lastCurrentState := currentState;
  IF result # error THEN
    currentState := saveState;
    result :=
      BuildAutomatonR
        (mode,
        .Operand2Of(automaton),
         tokenNumber,
         NDA,
         NDAActions,
         nextNewState,
         currentState);
   END;
  SetNextNDAStateSet
    (NDA, EPSILON, lastCurrentState, nextNewState);
  SetNextNDAStateSet
    (NDA, EPSILON, currentState, nextNewState);
  currentState := nextNewState;
  nextNewState := nextNewState + 1;
  IF mode = CONSTRUCT THEN
    SetAction
      (NDAActions, EPSILON, currentState, tokenNumber);
   END;
 ELSIF ClosureExpressionQ(automaton) THEN
  saveState := currentState;
  result :=
    BuildAutomatonR
      (mode,
```

```
        tokenStr,
        nextNewState,
        currentState);
  IF result = ok THEN
    result :=
      BuildAutomatonR
        (mode,
         fa,
         Operand2Of(re),
         subReList,
         tokenStr,
         nextNewState,
         currentState);
   END;
 ELSIF AlternationExpressionQ(re) THEN
      (* match either subexpression *)
  saveState := currentState;
  result :=
    BuildAutomatonR
      (mode,
       fa,
       Operand1Of(re),
       subReList,
       tokenStr,
       nextNewState,
       currentState);
  lastCurrentState := currentState;
  IF result = ok THEN
    currentState := saveState;
    result :=
      BuildAutomatonR
        (mode,
         fa,
         Operand2Of(re),
         subReList,
         tokenStr,
         nextNewState,
         currentState);
   END;
  AddState (fa, nextNewState, tokenStr, mode);
  EpsilonMove (fa, lastCurrentState, nextNewState);
  EpsilonMove (fa, currentState, nextNewState);
  currentState := nextNewState;
  nextNewState := nextNewState + 1;
 ELSIF ClosureExpressionQ(re) THEN
  saveState := currentState;
  result :=
    BuildAutomatonR
      (mode,
       fa,
       OperandOf(re),
       subReList,
         OperandOf(automaton),
         tokenNumber,
         NDA,
         NDAActions,
         nextNewState,
         currentState);
    SetNextNDAStateSet
      (NDA, EPSILON, currentState, saveState);
    currentState := saveState;
    IF mode = CONSTRUCT THEN
      SetAction
        (NDAActions, EPSILON, currentState, tokenNumber);
     END;
   ELSIF BrackettedExpressionQ(automaton) THEN
    result :=
      BuildAutomatonR
        (mode,
         OperandOf(automaton),
         tokenNumber,
         NDA,
         NDAActions,
         nextNewState,
         currentState);
   ELSIF TerminalQ(automaton) OR OneCharacterQ(automaton)
         OR CharacterStringQ(automaton) THEN
    result :=
      BuildAutomatonR
        (mode,
         automaton,
         tokenNumber,
         NDA,
         NDAActions,
         nextNewState,
         currentState);
   ELSIF LexicalNonTerminalQ(automaton) THEN
    result :=
      BuildAutomatonR
        (mode,
         automaton,
         tokenNumber,
         NDA,
         NDAActions,
         nextNewState,
         currentState);
   ELSE Error('BuildAutomaton: incorrect node type.');
   END;
  RETURN (result);
END BuildAutomaton;

PROCEDURE BuildAutomatonR
            (m : Mode;
```

```
         tokenStr,
         nextNewState,
         currentState);
    AddState (fa, nextNewState, tokenStr, mode);
    EpsilonMove (fa, saveState, nextNewState);
    EpsilonMove (fa, currentState, saveState);
    currentState := nextNewState;
    nextNewState := nextNewState + 1;
   ELSIF BrackettedExpressionQ(re) THEN
    result :=
      BuildAutomatonR
        (mode,
         fa,
         OperandOf(re),
         subReList,
         tokenStr,
         nextNewState,
         currentState);
   ELSIF TerminalQ(re) THEN
    result :=
      BuildAutomatonR
        (mode,
         fa,
         re,
         subReList,
         tokenStr,
         nextNewState,
         currentState);
   ELSIF LexicalNonTerminalQ(re) THEN
    result :=
      BuildAutomatonR
        (mode,
         fa,
         re,
         subReList,
         tokenStr,
         nextNewState,
         currentState);
   ELSE Error('BuildAutomaton: incorrect node type.');
   END;
  RETURN (result);
END BuildAutomaton;

PROCEDURE BuildAutomatonR
            (mode : Mode;
              automaton : Node;
              tokenNumber : CARDINAL;
              NDA : NDATable;
              NDAActions : ActionTable;
              VAR nextNewState, currentState : CARDINAL) :
             Result;

  VAR
    result : Result;
    i, j, saveState, lastCurrentState : CARDINAL;
    nextStateSet : Set;
    str : StringType;
    n : Node;

BEGIN
  IF ConcatenationExpressionQ(automaton) THEN
    result :=
      BuildAutomatonR
        (STEP,
         Operand1Of(automaton),
         tokenNumber,
         NDA,
         NDAActions,
         nextNewState,
         currentState);
    IF result = ok THEN
      result :=
        BuildAutomatonR
          (m,
           Operand2Of(automaton),
           tokenNumber,
           NDA,
           NDAActions,
           nextNewState,
           currentState);
     END;
    RETURN (result);
   ELSIF AlternationExpressionQ(automaton) THEN
    saveState := currentState;
    result :=
      BuildAutomatonR
        (m,
         Operand1Of(automaton),
         tokenNumber,
         NDA,
         NDAActions,
         nextNewState,
         currentState);
    lastCurrentState := currentState;
    IF result # error THEN
      currentState := saveState;
      result :=
```

```
                fa, re, subReList : Node;
                tokenStr : StringType;
                VAR nextNewState, currentState : CARDINAL) :
               Result;

  VAR
    result : Result;
    i, j, saveState, lastCurrentState : CARDINAL;
    tmpStr, stateStr, nextStateStr, str : StringType;
    n, chList : Node;

BEGIN
  IF ConcatenationExpressionQ(re) THEN
    result :=
      BuildAutomatonR
        (STEP,
         fa,
         Operand1Of(re),
         subReList,
         tokenStr,
         nextNewState,
         currentState);
    IF result = ok THEN
      result :=
        BuildAutomatonR
          (mode,
           fa,
           Operand2Of(re),
           subReList,
           tokenStr,
           nextNewState,
           currentState);
     END;
   ELSIF AlternationExpressionQ(re) THEN
    saveState := currentState;
    result :=
      BuildAutomatonR
        (mode,
         fa,
         Operand1Of(re),
         subReList,
         tokenStr,
         nextNewState,
         currentState);
    lastCurrentState := currentState;
    IF result = ok THEN
      currentState := saveState;
      result :=
        BuildAutomatonR
          (mode,
           fa,
           Operand2Of(re),
     BuildAutomatonR
       (m,
        Operand2Of(automaton),
        tokenNumber,
        NDA,
        NDAActions,
        nextNewState,
        currentState);
  END;
 SetNextNDAStateSet
   (NDA, EPSILON, lastCurrentState, nextNewState);
 SetNextNDAStateSet
   (NDA, EPSILON, currentState, nextNewState);
 currentState := nextNewState;
 nextNewState := nextNewState + 1;
 IF m = CONSTRUCT THEN
   SetAction
     (NDAActions, EPSILON, currentState, tokenNumber);
  END;
 RETURN (result);
ELSIF ClosureExpressionQ(automaton) THEN
 saveState := currentState;
 result :=
   BuildAutomatonR
     (m,
      OperandOf(automaton),
      tokenNumber,
      NDA,
      NDAActions,
      nextNewState,
      currentState);
 SetNextNDAStateSet
   (NDA, EPSILON, currentState, saveState);
 currentState := saveState;
 IF m = CONSTRUCT THEN
   SetAction
     (NDAActions, EPSILON, currentState, tokenNumber);
  END;
 RETURN (result);
ELSIF BrackettedExpressionQ(automaton) THEN
 result :=
   BuildAutomatonR
     (m,
      OperandOf(automaton),
      tokenNumber,
      NDA,
      NDAActions,
      nextNewState,
      currentState);
 RETURN (result);
ELSIF TerminalQ(automaton) OR OneCharacterQ(automaton)
      OR CharacterStringQ(automaton) THEN
        subReList,
        tokenStr,
        nextNewState,
        currentState);
   AddState (fa, nextNewState, tokenStr, mode);
   EpsilonMove (fa, lastCurrentState, nextNewState);
   EpsilonMove (fa, currentState, nextNewState);
   currentState := nextNewState;
   nextNewState := nextNewState + 1;
  END;
ELSIF ClosureExpressionQ(re) THEN
 saveState := currentState;
 result :=
   BuildAutomatonR
     (mode,
      fa,
      OperandOf(re),
      subReList,
      tokenStr,
      nextNewState,
      currentState);
 AddState (fa, nextNewState, tokenStr, mode);
 EpsilonMove (fa, saveState, nextNewState);
 EpsilonMove (fa, currentState, saveState);
 currentState := nextNewState;
 nextNewState := nextNewState + 1;
ELSIF BrackettedExpressionQ(re) THEN
 result :=
   BuildAutomatonR
     (mode,
      fa,
      OperandOf(re),
      subReList,
      tokenStr,
      nextNewState,
      currentState);
ELSIF TerminalQ(re) THEN
 RetrieveTerminalOf(re, str);
 tmpStr[2] := CHR(0);
 FOR i := 1 TO Length(str) DO
   tmpStr[1] := str[i];
   AddState (fa, nextNewState, tokenStr, mode);
   chList := MakeCharacterList();
   AppendNodeToList
     (chList, MakePrintingCharacter(tmpStr));
   AppendNodeToList
     (TransitionsOf
        (NthElement(currentState, StateListOf(fa))),
      MakeTransition
        (chList, CreateNumberNode(nextNewState)));
   currentState := nextNewState;
   nextNewState := nextNewState + 1;
 IF OneCharacterQ(automaton) THEN
   RetrieveOneCharacterOf(automaton, str);
  ELSE RetrieveCharacterStringOf(automaton, str);
  END;
 FOR i := 1 TO Length(str) DO
   nextStateSet :=
     GetNextNDAStateSet(NDA, str[i], currentState);
   IF EmptySetQ(nextStateSet) THEN
     SetNextNDAStateSet
       (NDA, str[i], currentState, nextNewState);
     currentState := nextNewState;
     nextNewState := nextNewState + 1;
    ELSE Error('BuildAutomatonR: collision.');
    END;
  END;
 IF m = CONSTRUCT THEN
   IF GetAction(NDAActions, str[i], currentState) =
          ERROR THEN
     SetAction
       (NDAActions, str[i], currentState, tokenNumber);
    ELSE RETURN (error);
    END;
  END;
 RETURN (ok);
ELSIF LexicalNonTerminalQ(automaton) THEN
 RetrieveIdentifierOf(IdentifierOf(automaton), str);
 n := FindSubAutomaton(str);
 IF LexicalClassRuleQ(n) THEN
   FOR i := 1 TO NumberOfComponentsOf(MembersOf(n)) DO
     RetrieveOneCharacterOf
       (GetComponentK(MembersOf(n), i), str);
     nextStateSet :=
       GetNextNDAStateSet(NDA, str[1], currentState);
     IF EmptySetQ(nextStateSet) THEN
       SetNextNDAStateSet
         (NDA, str[1], currentState, nextNewState);
       IF m = CONSTRUCT THEN
         SetAction
           (NDAActions,
            str[1],
            nextNewState,
            tokenNumber);
        END;
      ELSE Error('BuildAutomatonR: collision.');
      END;
    END;
   currentState := nextNewState;
   nextNewState := nextNewState + 1;
   RETURN (ok);
  ELSIF LexicalRuleQ(n) THEN
   result :=
     BuildAutomatonR
     END;
    result := ok;
   ELSIF LexicalNonTerminalQ(re) THEN
    RetrieveIdentifierOf(IdentifierOf(re), str);
    n := FindSubAutomaton(str, subReList);
    IF SubDefinitionQ(n) THEN
      result :=
        BuildAutomatonR
          (mode,
           fa,
           DefinitionOf(n),
           subReList,
           tokenStr,
           nextNewState,
           currentState);
     ELSIF ClassDefinitionQ(n) THEN
      AddState (fa, nextNewState, tokenStr, mode);
      chList := MakeCharacterList();
      FOR i := 1 TO ListLength(DefinitionOf(n)) DO
        RetrieveTerminalOf
          (NthElement(i, DefinitionOf(n)), str);
        IF Length(str) > 1 THEN
          Error
            (
         "BuildAutomatonR: ClassDefinition > 1 character."
             );
         END;
        AppendNodeToList
          (chList, MakePrintingCharacter(str));
       END;
      AppendNodeToList
        (TransitionsOf
           (NthElement(currentState, StateListOf(fa))),
         MakeTransition
           (chList, CreateNumberNode(nextNewState)));
      currentState := nextNewState;
      nextNewState := nextNewState + 1;
      result := ok;
     ELSE
      Error
        ('BuildAutomatonR: Unknown SubDefinition Type.');
     END;
   ELSE Error('BuildAutomatonR: incorrect node type.');
   END;
  RETURN (result);
END BuildAutomatonR;


PROCEDURE GetNextNFAEpsilonStateSet
            (nFA : Node;
             state : CARDINAL;
             VAR nextStateSet : Set);
          (m,
           DefinitionOf(n),
           tokenNumber,
           NDA,
           NDAActions,
           nextNewState,
           currentState);
      RETURN (result);
     ELSE
      Error('BuildAutomatonR: incorrect lexical re node.');
     END;
   ELSE Error('BuildAutomaton: incorrect node type.');
   END;
END BuildAutomatonR;


PROCEDURE MakeDATable () : DATable;

  VAR i, j : CARDINAL; t : DATable;

BEGIN
  ALLOCATE(t, SIZE(DATableRecord));
  FOR i := STARTCHARS TO MAXCHARS DO
    FOR j := START_STATE TO MAX_STATES DO
      t^.matrix[i, j] := UNASSIGNED;
     END;
   END;
  RETURN (t);
END MakeDATable;


PROCEDURE MakeActionTable () : ActionTable;

  VAR i, j : CARDINAL; t : ActionTable;

BEGIN
  ALLOCATE(t, SIZE(ActionTableRecord));
  FOR i := STARTCHARS TO MAXCHARS DO
    FOR j := START_STATE TO MAX_STATES DO
      t^.matrix[i, j] := ERROR;
     END;
   END;
  RETURN (t);
END MakeActionTable;


PROCEDURE MakeNDATable () : NDATable;

  VAR i, j : CARDINAL; t : NDATable;

BEGIN
  ALLOCATE(t, SIZE(NDATableRecord));
```

```
 VAR
   transitions, transition, ch : Node;
   nextStateStr : StringType;
   i, j, nextState : CARDINAL;
   okay : BOOLEAN;

 ClearSet(nextStateSet);
 transitions :=
   TransitionsOf(NthElement(state, StateListOf(nFA)));
 FOR i := 1 TO ListLength(transitions) DO
   transition := NthElement(i, transitions);
   FOR j := 1 TO ListLength(CharacterListOf(transition))
   DO
     ch := NthElement(j, CharacterListOf(transition));
     IF epsilonQ(ch) THEN
       RetrieveNumberOf
         (NextStateOf(transition), nextStateStr);
       StrToCard(nextStateStr, nextState, okay);
       IF NOT okay THEN
         Error
           ("GetNextNFAEpsilonStateSet: conversion error"
            );
        END;
       EnterKinSet(nextState, nextStateSet);
      END;
    END;
  END;
END GetNextNFAEpsilonStateSet;
```

```
  FOR i := STARTCHARS TO MAXCHARS DO
    FOR j := START_STATE TO MAX_STATES DO
      t^.matrix[i, j] := MakeSet();
     END;
   END;
  RETURN (t);
END MakeNDATable;


PROCEDURE FreeDATable (VAR t : DATable);

BEGIN
  IF t = NIL THEN
    Error("FreeDATable: table doesn't exist.");
   END;
  DEALLOCATE(t, SIZE(DATableRecord));
  t := NIL;
END FreeDATable;


PROCEDURE FreeActionTable (VAR t : ActionTable);

BEGIN
  IF t = NIL THEN
    Error("FreeActionTable: table doesn't exist.");
   END;
  DEALLOCATE(t, SIZE(ActionTableRecord));
  t := NIL;
END FreeActionTable;


PROCEDURE FreeNDATable (VAR t : NDATable);

  VAR i, j : CARDINAL;

BEGIN
  IF t = NIL THEN
    Error("FreeNDATable: table doesn't exist.");
   END;
  FOR i := STARTCHARS TO MAXCHARS DO
    FOR j := START_STATE TO MAX_STATES DO
      FreeSet(t^.matrix[i, j]);
     END;
   END;
  DEALLOCATE(t, SIZE(NDATableRecord));
  t := NIL;
END FreeNDATable;


PROCEDURE GetNextDAState
             c : CHAR;
             currentState : CARDINAL) : CARDINAL;
```

```
BEGIN
  IF t = NIL THEN Error('GetNextDAState: no table.'); END;
  RETURN (t^.matrix[ORD(c), currentState]);
END GetNextDAState;


PROCEDURE SetNextDAState
             (t : DATable;
              c : CHAR;
              currentState, nextState : CARDINAL);

BEGIN
  IF t = NIL THEN Error('SetNextDAState: no table.'); END;
  t^.matrix[ORD(c), currentState] := nextState;
END SetNextDAState;


PROCEDURE GetAction
             (t : ActionTable;
              c : CHAR;
              currentState : CARDINAL) : CARDINAL;

BEGIN
  IF t = NIL THEN Error('GetAction: no table.'); END;
  RETURN (t^.matrix[ORD(c), currentState]);
END GetAction;


PROCEDURE SetAction
             (t : ActionTable;
              c : CHAR;
              currentState, action : CARDINAL);

BEGIN
  IF t = NIL THEN Error('SetAction: no table.'); END;
  t^.matrix[ORD(c), currentState] := action;
END SetAction;


PROCEDURE GetNextNDAStateSet
             (t : NDATable;
              c : CHAR;
              currentState : CARDINAL) : Set;

BEGIN
  IF t = NDATable(NIL) THEN
    Error('GetNextNDAStateSEt: no table.');
   END;
  RETURN (t^.matrix[ORD(c), currentState]);
END GetNextNDAStateSet;
```

```
PROCEDURE SetNextNDAStateSet
            (t : NDATable;
             c : CHAR;
             currentState, nextState : CARDINAL);

BEGIN
  IF t = NDATable(NIL) THEN
    Error('SetNextNDAStateSEt: no table.');
   END;
  EnterKinSet(nextState, t^.matrix[ORD(c), currentState]);
END SetNextNDAStateSet;


PROCEDURE FillInNDATables
            (NDA : NDATable;
             NDAActions : ActionTable;
             sL : Symbols);

  VAR
    i, j : CARDINAL;
    str : StringType;
    currentState, nextNewState : CARDINAL;
    nextStateSet : Set;
    automaton : Node;
    result : Result;

BEGIN
  nextNewState := START_STATE + 1;
  FOR i := GetFirstTokenNumber(sL) TO
      GetLastTokenNumber(sL) DO
    currentState := START_STATE;
    SetNextNDAStateSet
      (NDA, EPSILON, currentState, nextNewState);
    currentState := nextNewState;
    nextNewState := nextNewState + 1;
    GetTokenString(i, str, sL);
    FOR j := 1 TO Length(str) DO
      nextStateSet :=
        GetNextNDAStateSet(NDA, str[j], currentState);
      IF EmptySetQ(nextStateSet) THEN
        SetNextNDAStateSet
          (NDA, str[j], currentState, nextNewState);
        currentState := nextNewState;
        nextNewState := nextNewState + 1;
       ELSE
        Error
          (
        'FillInNDATables: nonedeterministic string error.'
           );
       END;
     END;
```

```
      IF GetAction(NDAActions, str[j], currentState) =
             ERROR THEN
        SetAction(NDAActions, str[j], currentState, i);
       ELSE Error('FillInNDATables: Ambiguous tokens. ');
       END;
      nextStateSet :=
        GetNextNDAStateSet(NDA, ' ', currentState);
      IF EmptySetQ(nextStateSet) THEN
        SetNextNDAStateSet
          (NDA, ' ', currentState, nextNewState);
        currentState := nextNewState;
        nextNewState := nextNewState + 1;
       ELSE
        Error
          (
          'FillInNDATables: nonedeterministic string error.'
           );
       END;
      IF GetAction(NDAActions, ' ', currentState) =
             ERROR THEN
        SetAction(NDAActions, ' ', currentState, i + STOP);
       ELSE Error('FillInNDATables: Ambiguous tokens. ');
       END;
     END;
    FOR i := GetFirstAutomatonNumber(sL) TO
        GetLastAutomatonNumber(sL) DO
      automaton := GetAutomaton(i, sL);
      result :=
        BuildAutomaton
          (automaton, i, NDA, NDAActions, nextNewState);
      IF result = error THEN
        Error('FillInNDATables: automaton ambiguity.');
       END;
     END;
  END FillInNDATables;
```

```
  PROCEDURE EClosure (stateSet : Set; NDA : NDATable) : Set;

    VAR
      i, state : CARDINAL;
      eClosure, nextStateSet : Set;
      stack : Stack;
      setState : SetState;
      done : BOOLEAN;

  BEGIN
    setState := MakeSetState();
    stack := MakeStack();
    eClosure := MakeSet();
    i := First(stateSet, setState, done);
    WHILE NOT done DO
```

```
  PROCEDURE EClosure (stateSet : Set; nFA : Node) : Set;

    VAR
      i, nextState, state, k, m : CARDINAL;
      eClosure : Set;
      setState : SetState;
      done : BOOLEAN;
      chList, ch, transitions, transition : Node;
      nextStateStr : StringType;

  BEGIN
    setState := MakeSetState();
    eClosure := MakeSet();
    Union(eClosure, stateSet);
    state := First(stateSet, setState, done);
```

```
      EnterKinSet(i, eClosure);
      StackPush(i, stack);
      i := Next(setState, done);
    END;
  WHILE StackNotEmptyQ(stack) DO
    state := StackPop(stack);
    nextStateSet :=
      GetNextNDAStateSet(NDA, EPSILON, state);
    i := First(nextStateSet, setState, done);
    WHILE NOT done DO
      IF NOT MemberKQ(i, eClosure) THEN
        EnterKinSet(i, eClosure); StackPush(i, stack);
      END;
      i := Next(setState, done);
    END;
  END;
  FreeSetState(setState);
  RETURN (eClosure);
END EClosure;


PROCEDURE ConvertNDAtoDA
            (NDA : NDATable;
             NDAActions : ActionTable;
             DA : DATable;
             DAActions : ActionTable);

  TYPE SetArray = ARRAY [0..MAX_STATES] OF Set;

  VAR
    DStates : SetArray;
    DStatesNotMarked : Set;
    i, j, action, newAction, currentDState, foundDState,
      nextNewDState : CARDINAL;
    stateSet1, stateSet : Set;
    setState : SetState;
    unMarkedFlag, notFound, done : BOOLEAN;

BEGIN
  setState := MakeSetState();
  stateSet := MakeSet();
  DStatesNotMarked := MakeSet();
```

```
  DeleteKfromSet(state, stateSet);
  WHILE NOT done DO
    transitions :=
      TransitionsOf(NthElement(state, StateListOf(nFA)));
    FOR k := 1 TO ListLength(transitions) DO
      transition := NthElement(k, transitions);
      chList := CharacterListOf(transition);
      FOR m := 1 TO ListLength(chList) DO
        ch := NthElement(m, chList);
        IF epsilonQ(ch) THEN
          RetrieveNumberOf
            (NextStateOf(transition), nextStateStr);
          StrToCard(nextStateStr, nextState, okay);
          IF NOT okay THEN
            Error("EClosure: conversion error");
          END;
          IF NOT MemberKQ(nextState, eClosure) THEN
            EnterKinSet(nextState, eClosure);
            EnterKinSet(nextState, stateSet);
          END;
        END;
      END;
    END;
    state := First(stateSet, setState, done);
    DeleteKfromSet(state, stateSet);
  END;
  FreeSetState(setState);
  RETURN (eClosure);
END EClosure;


PROCEDURE ConvertNDAtoDA (nFA, dFA : Node);

  TYPE SetArray = ARRAY [0..MAX_STATES] OF Set;

  VAR
    DStates : SetArray;
    DStatesNotMarked : Set;
    state, i, j, k, m, output, newOutput, currentDState,
      foundDState, nextNewDState, nextState : CARDINAL;
    stateSet1, stateSet : Set;
    setState : SetState;
    ok, unMarkedFlag, notFound, found, done : BOOLEAN;
    outputStr, str, stateStr, chrStr, nextStateStr :
      StringType;
    chList, ch, transition, transitions : Node;

BEGIN
  chrStr[2] := CHR(0);
  setState := MakeSetState();
  stateSet := MakeSet();
  DStatesNotMarked := MakeSet();
```

```
DStates[FAIL] := MakeSet();
currentDState := START_STATE;
nextNewDState := currentDState + 1;
EnterKinSet(currentDState, stateSet);
DStates[currentDState] := EClosure(stateSet, NDA);
EnterKinSet(currentDState, DStatesNotMarked);
unMarkedFlag := TRUE;
WHILE unMarkedFlag DO (* mark current state (x) *)
  DeleteKfromSet(currentDState, DStatesNotMarked);
  FOR i := STARTCHARS TO MAXCHARS - 1 DO
    (* on 'a' from some state in the current state. *)
    ClearSet(stateSet);
    j := First(DStates[currentDState], setState, done);
    WHILE NOT done DO
      Union
        (stateSet, GetNextNDAStateSet(NDA, CHR(i), j));
      j := Next(setState, done);
     END;
    stateSet1 := EClosure(stateSet, NDA);
    IF EmptySetQ(stateSet1) THEN foundDState := FAIL;
     ELSE
      notFound := TRUE;
      j := START_STATE;
      WHILE (j < nextNewDState) AND notFound DO
        IF IdenticalSetsQ(stateSet1, DStates[j]) THEN
          notFound := FALSE; foundDState := j;
         END;
        j := j + 1;
       END;
      IF notFound THEN (* add a new state to D *)
        DStates[nextNewDState] := stateSet1;
        foundDState := nextNewDState;
        EnterKinSet(foundDState, DStatesNotMarked);
        nextNewDState := nextNewDState + 1;
       END;
     END;
    IF GetNextDAState(DA, CHR(i), currentDState) =
           UNASSIGNED THEN
      SetNextDAState
        (DA, CHR(i), currentDState, foundDState);
      action := ERROR;
      j := First(stateSet1, setState, done);
      WHILE NOT done DO
        newAction := GetAction(NDAActions, CHR(i), j);
        IF newAction ≠ ERROR THEN
          IF (newAction < action) OR
                 (action = ERROR) THEN
            action := newAction;
           END;
         END;
        newAction := GetAction(NDAActions, EPSILON, j);
        IF newAction ≠ ERROR THEN
```

```
DStates[FAIL] := MakeSet();
currentDState := START_STATE;
nextNewDState := currentDState + 1;
EnterKinSet(currentDState, stateSet);
DStates[currentDState] := EClosure(stateSet, nFA);
EnterKinSet(currentDState, DStatesNotMarked);
unMarkedFlag := TRUE;
WHILE unMarkedFlag DO (* mark current state (x) *)
  DeleteKfromSet(currentDState, DStatesNotMarked);
  FOR i := STARTCHARS TO MAXCHARS - 1 DO
    (* on 'a' from some state in the current state. *)
    ClearSet(stateSet);
    j := First(DStates[currentDState], setState, done);
    WHILE NOT done DO
      transitions :=
        TransitionsOf(NthElement(j, StateListOf(nFA)));
      FOR k := 1 TO ListLength(transitions) DO
        transition := NthElement(k, transitions);
        chList := CharacterListOf(transition);
        FOR m := 1 TO ListLength(chList) DO
          ch := NthElement(m, chList);
          IF PrintingCharacterQ(ch) THEN
            RetrievePrintingCharacterOf(ch, str);
           IF CHR(i) = str[1] THEN
              RetrieveNumberOf
                (NextStateOf(transition), nextStateStr);
              StrToCard(nextStateStr, nextState, ok);
              IF NOT ok THEN
                Error
                  ("ConvertNDAtoDA: conversion error");
               END;
              EnterKinSet(nextState, stateSet);
             END;
           ELSIF NonPrintingCharacterQ(ch) THEN
           END;
         END;
       END;
      j := Next(setState, done);
     END;
    stateSet1 := EClosure(stateSet, nFA);
    IF EmptySetQ(stateSet1) THEN foundDState := FAIL;
     ELSE
      notFound := TRUE;
      j := START_STATE;
      WHILE (j < nextNewDState) AND notFound DO
       IF IdenticalSetsQ(stateSet1, DStates[j]) THEN
          notFound := FALSE; foundDState := j;
         END;
        j := j + 1;
       END;
      IF notFound THEN (* add a new state to D *)
        DStates[nextNewDState] := stateSet1;
```

```
            IF (newAction < action) OR
                   (action = ERROR) THEN
              action := newAction;
             END;
           END;
          j := Next(setState, done);
         END;
        SetAction(DAActions, CHR(i), foundDState, action);
       END;
     END;
    currentDState :=
      First(DStatesNotMarked, setState, done);
    IF done THEN unMarkedFlag := FALSE;
     ELSE unMarkedFlag := TRUE;
     END;
   END;
  FreeSetState(setState);
END ConvertNDAtoDA;


PROCEDURE MakeLexicalAnalyzer
            (grammarRuleList : Node; sL : Symbols) :
            LexicalAnalyzerTables;

  VAR
    NDA : NDATable;
    NDAActions : ActionTable;
    t : LexicalAnalyzerTables;

BEGIN
  t := MakeLexicalAnalyzerTables();
  NDA := MakeNDATable();
  NDAActions := MakeActionTable();
  FillInNDATables(NDA, NDAActions, sL);
  ConvertNDAtoDA
    (NDA, NDAActions, StateTableOf(t), ActionTableOf(t));
  FreeNDATable(NDA);
  FreeActionTable(NDAActions);
  RETURN (t);
END MakeLexicalAnalyzer;


PROCEDURE PrintDATable (t : DATable);

  VAR i, j, count : CARDINAL;

BEGIN
  WriteString(" Next State Table ");
  WriteString("     ");
  FOR i := STARTCHARS TO MAXCHARS - 1 DO
    Write(CHR(i)); WriteString(" ");
   END;
```

```
   j := First(stateSet1, setState, done);
    found := FALSE;
    output := ERROR;
    WHILE (NOT done) DO
      IF AcceptQ
               (TypeOf
                  (NthElement
                     (j, StateListOf(nFA)))) THEN
        found := TRUE;
        RetrieveNumberOf
          (OutputOf
             (TypeOf
                (NthElement(j, StateListOf(nFA)))),
           outputStr);
        StrToCard(outputStr, newOutput, ok);
        IF NOT ok THEN
          Error
            ("ConvertNDAtoDA: conversion error.");
         END;
        IF (output > newOutput) OR
               (output = ERROR) THEN
          output := newOutput;
         END;
       END;
      j := Next(setState, done);
     END;
    foundDState := nextNewDState;
    nextNewDState := nextNewDState + 1;
    EnterKinSet(foundDState, DStatesNotMarked);
    CardToStr(foundDState, stateStr, 0, okay);
    IF NOT okay THEN
      Error("ConvertNDAtoDA: conversion error.");
     END;
    IF found THEN
      CardToStr(output, outputStr, 0, okay);
      IF NOT okay THEN
        Error("ConvertNDAtoDA: conversion error.");
       END;
      AddState (dFA, stateStr, outputStr, CONSTRUCT);
     ELSE
      AddState (dFA, stateStr, "", STEP);
     END;
   END;
 END;
chrStr[1] := CHR(i);
transitions :=
  TransitionsOf
    (NthElement(currentDState, StateListOf(dFA)));
found := FALSE;
j := 1;
WHILE (NOT found) AND
      (j <= ListLength(transitions)) DO
  WriteString('e');
  WriteLn();
  FOR j := START_STATE TO MAX_STATES DO
    WriteCard(j, 0);
    WriteString(" ");
    count := 0;
    FOR i := STARTCHARS TO MAXCHARS DO
      WriteCard(t^.matrix[i, j], 0);
      WriteString(' ');
      IF count > 100 THEN
        count := 0; WriteLn(); WriteString("    ");
       ELSE count := count + 3;
       END;
     END;
    WriteLn();
   END;
END PrintDATable;


PROCEDURE PrintNDATable (t : NDATable);

  VAR i, j, count : CARDINAL;

BEGIN
  WriteString(" Next State Table ");
  WriteLn();
  WriteString("     ");
  FOR i := STARTCHARS TO MAXCHARS - 1 DO
    Write(CHR(i)); WriteString(" ");
   END;
  WriteString('e');
  WriteLn();
  FOR j := START_STATE TO MAX_STATES DO
    WriteCard(j, 0);
    WriteString(" ");
    count := 0;
    FOR i := STARTCHARS TO MAXCHARS DO
      PrintSet(t^.matrix[i, j]);
      WriteString(' ');
      IF count > 100 THEN
        count := 0; WriteLn(); WriteString("    ");
       ELSE count := count + 6;
       END;
     END;
    WriteLn();
   END;
END PrintNDATable;


PROCEDURE PrintActionTable (t : ActionTable);

  VAR i, j, count : CARDINAL;
```

```
          transition := NthElement(j, transitions);
          RetrieveNumberOf(NextStateOf(transition), str);
          StrToCard(str, state, ok);
          IF NOT okay THEN
            Error("ConvertNDAtoDA: conversion error.");
           END;
          IF state = foundDState THEN
            found := TRUE;
            IF state # FAIL THEN
              AppendNodeToList
                (CharacterListOf(transition),
                 MakePrintingCharacter(chrStr));
             END;
           END;
          j := j + 1;
         END;
        IF (NOT found) AND (foundDState # FAIL) THEN
          CardToStr(foundDState, stateStr, 0, okay);
          IF NOT okay THEN
            Error("ConvertNDAtoDA: conversion error.");
           END;
          chList := MakeCharacterList();
          AppendNodeToList
            (chList, MakePrintingCharacter(chrStr));
          AppendNodeToList
            (TransitionsOf
               (NthElement
                  (currentDState, StateListOf(dFA))),
             MakeTransition(chList, MakeNumber(stateStr)));
         END;
       END;
      currentDState :=
        First(DStatesNotMarked, setState, done);
      IF done THEN unMarkedFlag := FALSE;
       ELSE unMarkedFlag := TRUE;
       END;
     END;
    FreeSet(stateSet);
    FreeSet(DStatesNotMarked);
  END ConvertNDAtoDA;

BEGIN
  WriteString("Initialization.");
  WriteLn();
  CardToStr(ERROR, errorStr, 0, okay);
  IF NOT okay THEN
    Error("BuildAutomaton: conversion error.");
   END;
  argc := NoOfArguments();
  IF argc < 2 THEN
    WriteString("Usage: lex InputFile."); WriteLn();
   ELSE
  BEGIN
    WriteString(" Action Table ");
    WriteLn();
    WriteString("      ");
    FOR i := STARTCHARS TO MAXCHARS DO
      Write(CHR(i)); WriteString(" ");
     END;
    WriteString('e');
    WriteLn();
    FOR j := START_STATE TO MAX_STATES DO
      WriteCard(j, O);
      WriteString(" ");
      count := O;
      FOR i := STARTCHARS TO MAXCHARS DO
        WriteCard(t^.matrix[i, j], O);
        WriteString(' ');
        IF count > 100 THEN
          count := O; WriteLn(); WriteString("    ");
         ELSE count := count + 3;
         END;
       END;
      WriteLn();
     END;
  END PrintActionTable;

END LexGen.
```

```
FOR i := 1 TO argc - 1 DO
  GetArgument(i, arg, upResult);
  IF (upResult = upNotFound) THEN
    WriteString("lex: Command Line Argument Problems.");
    WriteLn();
    HALT();
   END;
  Assign(arg, fileName, okay);
  IF NOT okay THEN
    WriteString("lex: assignment error.");
    WriteLn();
    HALT();
   END;
 END;
WriteString("Parsing Input.");
WriteLn();
res :=
  ParseInputFile
    (fileName, "REGrammar", "REInput", okay);
IF NOT okay THEN
  WriteString("Parse Failed."); WriteLn();
 ELSE
  WriteString("Constructing NDFA.");
  WriteLn();
  tokenNumber := O;
  stateNumber := START_STATE;
  reList := REListOf(res);
  reListLen := ListLength(reList);
  subReList := SubREListOf(res);
  fa := MakeFA(MakeStateList());
  CardToStr(FAIL, failStr, O, okay);
  IF NOT okay THEN
    Error("lex: conversion error.");
   END;
  AddState (fa, FAIL, "", STEP);
  AddState (fa, START_STATE, "", STEP);
  stateNumber := stateNumber + 1;
  FOR i := 1 TO reListLen DO
    re := NthElement(i, reList);
    tokenNumber := tokenNumber + 1;
    result :=
      BuildAutomaton
        (fa, re, subReList, tokenNumber, stateNumber);
   END;
  WriteString("Converting NDFA to DFA.");
  WriteLn();
  dFA := MakeFA(MakeStateList());
  AddState (dFA, FAIL, "", STEP);
  AddState (dFA, START_STATE, "", STEP);
  ConvertNDAtoDA(fa, dFA);
  WriteString("Writing DFA Tables.");
  WriteLn();
```

```
      Concat(fileName, "_DFATables", name, okay);
      IF NOT okay THEN Error("lex: concatenation error.");
       END;
      SaveParseTree(dFA, name);
     END;
   END;
END lexProgram.
```

6 Lexical Analyzer Comparison

This section contains the source code for both the old and new versions of the lexical analyzer. The old version (used in GRAFS) is on the left and the new version (which uses GRAFS) is on the right. Note that we have attempted to align the corresponding sections of code to make comparison easier.

```
IMPLEMENTATION MODULE LexAnalyzer;

  FROM LexGen IMPORT
    DATable, ActionTable, START_STATE, GetNextDAState,
    GetAction, FAIL, STOP;
  FROM GrammarSymbols IMPORT ERROR, EOF, START_TOKEN;
  FROM DataTypes IMPORT StringType;
  FROM ErrorModule IMPORT Error;
```

```
MODULE lexAnalyzerProgram;

  FROM ErrorModule IMPORT Error;
  FROM DataTypes IMPORT Node, StringType;
  FROM InOut IMPORT WriteString, WriteLn, WriteCard;
  FROM UnixParam IMPORT
    UPResult, NoOfArguments, GetArgument;
  FROM String IMPORT
    Compare, CompareResult, Length, Assign, Concat;
  FROM Storage IMPORT ALLOCATE, DEALLOCATE;
  FROM Convert IMPORT StrToCard, CardToStr;
  FROM StackModule IMPORT
    Stack, StackPush, StackPop, StackNotEmptyQ, MakeStack,
    FreeStack;
  FROM SetModule IMPORT
    Set, MakeSet, FreeSet, EnterKinSet, DeleteKfromSet,
    MemberKQ, IdenticalSetsQ, Union, ClearSet, PrintSet,
    EmptySetQ, SetState, MakeSetState, FreeSetState, First,
    Next;
  FROM General IMPORT
    ParseInputFile, PrettyPrint, UnParse, NthElement,
    ListLength, AppendNodeToList, EmptyNodeQ, SaveParseTree,
    RecoverParseTree;
  FROM FAGrammar_GS IMPORT
    MakeFA, MakeStateList, MakeState, MakeTransitionList,
    MakeTransition, MakeAccept, MakeReject, Makeepsilon,
    Makeeoln, Makeeof, MakeCharacterList, MakeNumber,
    MakePrintingCharacter, FAQ, StateListQ, StateQ,
    TransitionListQ, TransitionQ, TypeQ, AcceptQ, RejectQ,
    CharacterQ, NonPrintingCharacterQ, epsilonQ, eolnQ,
    eofQ, NumberQ, PrintingCharacterQ, StateListOf,
    NumberOf, TransitionsOf, TypeOf, CharacterListOf,
    OutputOf, NextStateOf, RetrieveNumberOf,
    RetrievePrintingCharacterOf, MakeDefault, DefaultOf;
  FROM SimpleIO IMPORT ReadChar, ReadLn, EOL, EOT;
```

```
CONST MAX_BUFFER_ADDRESS = 255;

TYPE Buffer = ARRAY [0..MAX_BUFFER_ADDRESS] OF CHAR;

VAR
  EOFlag : BOOLEAN;
  buffer : Buffer;
  head, tail : CARDINAL;
  lookAhead : CARDINAL;
  MetaFlag, stopFlag : BOOLEAN;
  tokenLine, tokenColumn : CARDINAL;
  lastTokenLine, lastTokenColumn : CARDINAL;


PROCEDURE InitializeLexicalAnalyzer ();

BEGIN
  tokenLine := 1;
  tokenColumn := 1;
  lastTokenLine := 1;
  lastTokenColumn := 1;
END InitializeLexicalAnalyzer;


PROCEDURE GetTokenLine () : CARDINAL;

BEGIN
  RETURN (lastTokenLine);
END GetTokenLine;
```

```
CONST
  ERROR = 0;
  FAIL = 1;
  START_STATE = 2;
  STOP = 5000;
  EOF = 1;
  STARTCHARS = ORD(' ');
  MAXCHARS = ORD(' ');
  MAX_BUFFER_ADDRESS = 255;

TYPE
  Result = (ok, error);
  Mode = (STEP, CONSTRUCT);
  Buffer = ARRAY [0..MAX_BUFFER_ADDRESS] OF CHAR;
  InputProc = PROCEDURE () : CHAR;

VAR
  argc, i, token : CARDINAL;
  tableName, tokenStr, failStr, str, arg, name, fileName :
    StringType;
  upResult : UPResult;
  result : Result;
  okay : BOOLEAN;
  dFA : Node;
  EOFlag : BOOLEAN;
  buffer : Buffer;
  head, tail : CARDINAL;
  lookAhead : CARDINAL;
  MetaFlag, stopFlag : BOOLEAN;
  tokenLine, tokenColumn : CARDINAL;
  lastTokenLine, lastTokenColumn : CARDINAL;


PROCEDURE InitializeLexicalAnalyzer ();

BEGIN
  tokenLine := 1;
  tokenColumn := 1;
  lastTokenLine := 1;
  lastTokenColumn := 1;
END InitializeLexicalAnalyzer;


PROCEDURE GetTokenLine () : CARDINAL;

BEGIN
  RETURN (lastTokenLine);
END GetTokenLine;
```

```
PROCEDURE GetTokenColumn () : CARDINAL;

BEGIN
  RETURN (lastTokenColumn);
END GetTokenColumn;


PROCEDURE IncrementTokenLine ();

BEGIN
  tokenLine := tokenLine + 1; tokenColumn := 1;
END IncrementTokenLine;


PROCEDURE Eof (c : CHAR) : BOOLEAN;

BEGIN
  RETURN (ORD(c) = EOF);
END Eof;


PROCEDURE Delimiter
            (VAR c : CHAR; VAR MetaFlag : BOOLEAN) :
            BOOLEAN;

BEGIN
  IF MetaFlag THEN MetaFlag := FALSE; RETURN (FALSE); END;
  IF (c <= ' ') OR (c > CHR(127)) THEN RETURN (TRUE); END;
  RETURN (FALSE);
END Delimiter;


PROCEDURE EndDelimiter
            (VAR c : CHAR; VAR MetaFlag : BOOLEAN) :
            BOOLEAN;

BEGIN
  IF (c < ' ') OR (c > CHR(127)) THEN
    IF MetaFlag THEN MetaFlag := FALSE; RETURN (FALSE);
     ELSE RETURN (TRUE);
     END;
   END;
  RETURN (FALSE);
END EndDelimiter;


PROCEDURE NextToken
            (GetChar : InputProc;
             t : DATable;
             action : ActionTable;
```

```
PROCEDURE GetTokenColumn () : CARDINAL;

BEGIN
  RETURN (lastTokenColumn);
END GetTokenColumn;


PROCEDURE IncrementTokenLine ();

BEGIN
  tokenLine := tokenLine + 1; tokenColumn := 1;
END IncrementTokenLine;


PROCEDURE Eof (c : CHAR) : BOOLEAN;

BEGIN
  RETURN (ORD(c) = EOF);
END Eof;


PROCEDURE Delimiter
            (VAR c : CHAR; VAR MetaFlag : BOOLEAN) :
            BOOLEAN;

BEGIN
  IF MetaFlag THEN MetaFlag := FALSE; RETURN (FALSE); END;
 IF (c <= ' ') OR (c > CHR(127)) THEN RETURN (TRUE); END;
  RETURN (FALSE);
END Delimiter;


PROCEDURE EndDelimiter
            (VAR c : CHAR; VAR MetaFlag : BOOLEAN) :
            BOOLEAN;

BEGIN
 IF (c < ' ') OR (c > CHR(127)) THEN
    IF MetaFlag THEN MetaFlag := FALSE; RETURN (FALSE);
     ELSE RETURN (TRUE);
     END;
   END;
  RETURN (FALSE);
END EndDelimiter;


PROCEDURE NextToken
            (GetChar : InputProc;
             dFA : Node;
             VAR tokenString : StringType) : CARDINAL;
```

```
          VAR tokenString : StringType) : CARDINAL;

VAR
  c, c1 : CHAR;
  charCount : CARDINAL;
  currentState : CARDINAL;
  notDone : BODLEAN;
  a : CARDINAL;
  i : CARDINAL;
  tokenStringCount : CARDINAL;
  lastAction : CARDINAL;


PROCEDURE BufferEmptyQ () : BOOLEAN;

BEGIN
  RETURN
    ((head = (tail + 1)) OR
     ((head = 0) AND (tail = MAX_BUFFER_ADDRESS)));
END BufferEmptyQ;


PRDCEDURE ReSetBufferLookAhead ();

BEGIN
  IF tail = MAX_BUFFER_ADDRESS THEN lookAhead := 0;
   ELSE lookAhead := tail + 1;
   END;
END ReSetBufferLookAhead;


PROCEDURE ScanBufferChar () : CHAR;

  VAR c : CHAR;

BEGIN
  IF lookAhead = head THEN
    IF head = tail THEN
      Error("ScanBufferChar: buffer full.");
     END;
    buffer[head] := GetChar();
    IF head = MAX_BUFFER_ADDRESS THEN head := 0;
     ELSE head := head + 1;
     END;
    c := buffer[lookAhead];
    lookAhead := head;
   ELSE
    c := buffer[lookAhead];
    IF lookAhead = MAX_BUFFER_ADDRESS THEN
      lookAhead := 0;
```

```
VAR
  c, c1 : CHAR;
  charCount : CARDINAL;
  currentState : CARDINAL;
  notDone : BOOLEAN;
  a : CARDINAL;
  i : CARDINAL;
  tokenStringCount : CARDINAL;
  lastAction : CARDINAL;


PRDCEDURE BufferEmptyQ () : BDDLEAN;

BEGIN
  RETURN
    ((head = (tail + 1)) OR
     ((head = 0) AND (tail = MAX_BUFFER_ADDRESS)));
END BufferEmptyQ;


PRDCEDURE ReSetBufferLookAhead ();

BEGIN
  IF tail = MAX_BUFFER_ADDRESS THEN lookAhead := 0;
   ELSE lookAhead := tail + 1;
   END;
END ReSetBufferLookAhead;


PRDCEDURE ScanBufferChar () : CHAR;

  VAR c : CHAR;

BEGIN
  IF lookAhead = head THEN
    IF head = tail THEN
      Error("ScanBufferChar: buffer full.");
     END;
    buffer[head] := GetChar();
    IF head = MAX_BUFFER_ADDRESS THEN head := 0;
     ELSE head := head + 1;
     END;
    c := buffer[lookAhead];
    lookAhead := head;
   ELSE
    c := buffer[lookAhead];
    IF lookAhead = MAX_BUFFER_ADDRESS THEN
      lookAhead := 0;
     ELSE lookAhead := lookAhead + 1;
    ELSE lookAhead := lookAhead + 1;
    END;
  END;
  RETURN (c);
END ScanBufferChar;


PROCEDURE PutBufferChar (c : CHAR);

BEGIN
  buffer[head] := c;
  IF head = MAX_BUFFER_ADDRESS THEN head := 0;
   ELSE head := head + 1;
   END;
END PutBufferChar;


PROCEDURE GetBufferChar () : CHAR;

  VAR c : CHAR;

BEGIN
  IF BufferEmptyQ() THEN
    tokenColumn := tokenColumn + 1; RETURN (GetChar());
   END;
  IF tail = MAX_BUFFER_ADDRESS THEN tail := 0;
   ELSE tail := tail + 1;
   END;
  IF tail = lookAhead THEN
    IF lookAhead = MAX_BUFFER_ADDRESS THEN
      lookAhead := 0;
     ELSE lookAhead := lookAhead + 1;
     END;
   END;
  c := buffer[tail];
  tokenColumn := tokenColumn + 1;
  RETURN (c);
END GetBufferChar;
```

```
    END;
  END;
  RETURN (c);
END ScanBufferChar;


PROCEDURE PutBufferChar (c : CHAR);

BEGIN
  buffer[head] := c;
  IF head = MAX_BUFFER_ADDRESS THEN head := 0;
   ELSE head := head + 1;
   END;
END PutBufferChar;


PROCEDURE GetBufferChar () : CHAR;

  VAR c : CHAR;

BEGIN
  IF BufferEmptyQ() THEN
    tokenColumn := tokenColumn + 1; RETURN (GetChar());
   END;
  IF tail = MAX_BUFFER_ADDRESS THEN tail := 0;
   ELSE tail := tail + 1;
   END;
  IF tail = lookAhead THEN
    IF lookAhead = MAX_BUFFER_ADDRESS THEN
      lookAhead := 0;
     ELSE lookAhead := lookAhead + 1;
     END;
   END;
  c := buffer[tail];
  tokenColumn := tokenColumn + 1;
  RETURN (c);
END GetBufferChar;


PROCEDURE GetNextDFAState
            (dFA : Node; c : CHAR; state : CARDINAL) :
            CARDINAL;

  VAR
    transitions, transition, chList, ch : Node;
    i, j, cardinal : CARDINAL;
    ok : BOOLEAN;
    str : StringType;

BEGIN
```

```
  transitions :=
    TransitionsOf(NthElement(state, StateListOf(dFA)));
  FOR i := 1 TO ListLength(transitions) DO
    transition := NthElement(i, transitions);
    chList := CharacterListOf(transition);
    FOR j := 1 TO ListLength(chList) DO
      ch := NthElement(j, chList);
      IF PrintingCharacterQ(ch) THEN
        RetrievePrintingCharacterOf(ch, str);
        IF c = str[1] THEN
          RetrieveNumberOf
            (NextStateOf(transition), str);
          StrToCard(str, cardinal, ok);
          IF NOT ok THEN
            Error("GetOutput : Conversion error.");
           END;
          RETURN (cardinal);
         END;
      ELSIF eolnQ(ch) THEN
      ELSIF eofQ(ch) THEN
      ELSIF epsilonQ(ch) THEN
       Error("NextToken : Bad Table.");
      END;
    END;
   END;
  RetrieveNumberOf
    (NextStateOf
       (DefaultOf(NthElement(state, StateListOf(dFA)))),
     str);
  StrToCard(str, cardinal, ok);
  IF NOT ok THEN Error("GetOutput : Conversion error.");
   END;
  RETURN (cardinal);
END GetNextDFAState;


PROCEDURE GetOutput
            (dFA : Node; state : CARDINAL) : CARDINAL;

  VAR
    s : Node;
    ok : BOOLEAN;
    str : StringType;
    cardinal : CARDINAL;

BEGIN
  s := NthElement(state, StateListOf(dFA));
  IF AcceptQ(TypeOf(s)) THEN
    RetrieveNumberOf(OutputOf(TypeOf(s)), str);
    StrToCard(str, cardinal, ok);
```

```
BEGIN
  lastTokenColumn := tokenColumn;
  lastTokenLine := tokenLine;
  ReSetBufferLookAhead();
  IF EOFlag THEN RETURN (EOF); END;
  IF NOT Eof(c) THEN c := GetBufferChar(); END;
  IF c = '~' THEN MetaFlag := TRUE; c := GetBufferChar();
   END;
  WHILE NOT Eof(c) AND Delimiter(c, MetaFlag) DO
    c := GetBufferChar();
   END;
  IF NOT Eof(c) THEN
    PutBufferChar(c); c := ScanBufferChar();
   END;
  charCount := 0;
  tokenStringCount := 0;
  currentState := START_STATE;
  notDone := TRUE;
  lastAction := ERROR;
  IF c = '~' THEN MetaFlag := TRUE; c := ScanBufferChar();
   END;
  WHILE NOT Eof(c) AND NOT EndDelimiter(c, MetaFlag) AND
        notDone DO
    currentState := GetNextDAState(t, c, currentState);
    IF currentState # FAIL THEN
      charCount := charCount + 1;
      a := GetAction(action, c, currentState);
      IF a > STOP THEN
        a := a - STOP;
        IF MetaFlag THEN MetaFlag := FALSE;
         ELSE notDone := FALSE;
         END;
       END;
      IF a > ERROR THEN
        FOR i := 1 TO charCount DO
          tokenStringCount := tokenStringCount + 1;
          c1 := GetBufferChar();
          IF c1 = '~' THEN c1 := GetBufferChar(); END;
          tokenString[tokenStringCount] := c1;
         END;
        lastAction := a;
        charCount := 0;
```

```
      IF NOT ok THEN
        Error("GetOutput : Conversion error.");
       END;
      RETURN (cardinal);
    ELSE RETURN (ERROR);
    END;
  END GetOutput;

BEGIN
  lastTokenColumn := tokenColumn;
  lastTokenLine := tokenLine;
  ReSetBufferLookAhead();
  IF EOFlag THEN RETURN (EOF); END;
  IF NOT Eof(c) THEN c := GetBufferChar(); END;
  IF c = '~' THEN MetaFlag := TRUE; c := GetBufferChar();
   END;
  WHILE NOT Eof(c) AND Delimiter(c, MetaFlag) DO
    c := GetBufferChar();
   END;
  IF NOT Eof(c) THEN
    PutBufferChar(c); c := ScanBufferChar();
   END;
  charCount := 0;
  tokenStringCount := 0;
  currentState := START_STATE;
  notDone := TRUE;
  lastAction := ERROR;
  IF c = '~' THEN MetaFlag := TRUE; c := ScanBufferChar();
   END;
  WHILE NOT Eof(c) AND NOT EndDelimiter(c, MetaFlag) AND
        notDone DO
    currentState := GetNextDFAState(dFA, c, currentState);
    IF currentState # FAIL THEN
      charCount := charCount + 1;
      a := GetOutput(dFA, currentState);
      IF a > STOP THEN
        a := a - STOP;
        IF MetaFlag THEN MetaFlag := FALSE;
         ELSE notDone := FALSE;
         END;
       END;
      IF a > ERROR THEN
        FOR i := 1 TO charCount DO
          tokenStringCount := tokenStringCount + 1;
          c1 := GetBufferChar();
          IF c1 = '~' THEN c1 := GetBufferChar(); END;
          tokenString[tokenStringCount] := c1;
         END;
        lastAction := a;
        charCount := 0;
```

```
         END;
        c := ScanBufferChar();
        IF c = '`' THEN
          MetaFlag := TRUE; c := ScanBufferChar();
         END;
       ELSE notDone := FALSE;
       END;
     END;
    IF tokenStringCount > 0 THEN
      stopFlag := FALSE;
      WHILE (tokenString[tokenStringCount] = ' ') AND
            (NOT stopFlag) DO
        IF tokenStringCount > 1 THEN
          tokenStringCount := tokenStringCount - 1;
         ELSE stopFlag := TRUE;
         END;
       END;
     END;
    tokenString[tokenStringCount + 1] := CHR(0);
    IF Eof(c) AND BufferEmptyQ() THEN
      IF lastAction = ERROR THEN RETURN (EOF);
       ELSE; EOFlag := TRUE;
       END;
     END;
    RETURN (lastAction);
  END NextToken;

BEGIN
  head := 1;
  tail := 0;
  lookAhead := 1;
  EOFlag := FALSE;
  MetaFlag := FALSE;
END LexAnalyzer.
```

```
         END;
        c := ScanBufferChar();
        IF c = '`' THEN
          MetaFlag := TRUE; c := ScanBufferChar();
         END;
       ELSE notDone := FALSE;
       END;
     END;
    IF tokenStringCount > 0 THEN
      stopFlag := FALSE;
      WHILE (tokenString[tokenStringCount] = ' ') AND
            (NOT stopFlag) DO
        IF tokenStringCount > 1 THEN
          tokenStringCount := tokenStringCount - 1;
         ELSE stopFlag := TRUE;
         END;
       END;
     END;
    tokenString[tokenStringCount + 1] := CHR(0);
    IF Eof(c) AND BufferEmptyQ() THEN
      IF lastAction = ERROR THEN RETURN (EOF);
       ELSE; EOFlag := TRUE;
       END;
     END;
    RETURN (lastAction);
  END NextToken;


  PROCEDURE GetChar () : CHAR;

    VAR c : CHAR;

  BEGIN
    ReadChar(c);
    IF EOL() AND NOT EOT() THEN
      IncrementTokenLine();
      ReadLn();
      c := CHR(0);
      IF EOT() THEN RETURN (CHR(EOF)); ELSE RETURN (c); END;
     ELSIF EOT() THEN RETURN (CHR(EOF));
     ELSE RETURN (c);
     END;
  END GetChar;

BEGIN
  WriteString("Initialization.");
  WriteLn();
  head := 1;
  tail := 0;
  lookAhead := 1;
  EOFlag := FALSE;
  MetaFlag := FALSE;
  argc := NoOfArguments();
  IF argc < 2 THEN
    WriteString
      ("Usage: lex TableFile <InputFile >OutputFile.");
    WriteLn();
   ELSE
    FOR i := 1 TO argc - 1 DO
      GetArgument(i, arg, upResult);
      IF (upResult = upNotFound) THEN
        WriteString("lex: Command Line Argument Problems.");
        WriteLn();
        HALT();
       END;
      Assign(arg, tableName, okay);
      IF NOT okay THEN
        WriteString("lex: assignment error.");
        WriteLn();
        HALT();
       END;
     END;
    InitializeLexicalAnalyzer();
    WriteString("Recovering Tables.");
    WriteLn();
    dFA := RecoverParseTree(tableName);
    WriteString("Analysis.");
    WriteLn();
    token := NextToken(GetChar, dFA, tokenStr);
    WHILE token # ERROR DO
      WriteString("token: ");
      WriteCard(token, 0);
      WriteString("tokenString:");
      WriteString(tokenStr);
      WriteString(".");
      WriteLn();
      token := NextToken(GetChar, dFA, tokenStr);
     END;
   END;
END lexAnalyzerProgram.
```

REFERENCES

[AhoUll79]
Aho, A. V., Ullman, J. D., *Principles of Compiler Design*, Addison-Wesley, (1977).

[BaShSa84]
Barstow, Shrobe, and Sandewall, editors, *Interactive Programming Environments*, McGraw Hill, (1984).

[Cam86]
Cameron, Robert D., "Source Encoding using Syntactic Information Source Models", *LCCR Technical Report 86-7*, School of Computing Science, Simon Fraser University, (Sept. 1986), 24 pages.

[Cam87a]
Cameron, Robert D., "Prettyprinter Abstraction Using Procedural Parameters", *LCCR Technical Report 87-4*, School of Computing Science, Simon Fraser University, (Feb. 1987), 10 pages.

[Cam87b]
Cameron, Robert D., "Pascal MPS Manual", School of Computing Science, *draft* Simon Fraser University, (1987).

[CamIto84]
Cameron, Robert D. and Ito, M. R., "Grammar-Based Definition of Metaprogramming Systems", *ACM Transactions on Programming Languages and Systems*, 6-1, (Jan. 1984) pp. 20-54.

[Don83]
Donzeau-Gouge, V., Kahn, G., Lang, B., Melese, B., Morcos, E., "Outline of a Tool for Document Manipulation", *IFIP*, Paris, (Sept. 1983).

[Don84a]
Donzeau-Gouge, V., Kahn, G., Lang, B., Melese, B., "Document Structure and Modularity in Mentor", *SigPlan Notices*, 19-5, (May 1984) pp. 141-148.

[Don84b]
Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B., "Programming Environments Based on Structured Editors: The Mentor Experience", *Interactive Programming Environments*, ed. Barstow, Shrobe, and Sandewall, McGraw Hill, (1984).

[EhMa85]
Ehrig, H., Mahr, B., *Fundamentals of Algebraic Specifications 1*, Springer-Verlag, (1985).

[GheJaz82]
Ghezzi, C., Jazayeri, M., *Programming Language Concepts*, John Wiley and Sons, Inc., (1982).

[Go84]
Gorman, Michael M., *Managing Data Base: Four Critical Factors*, QED Information Sciences Inc., (1984).

[HeeKli85]
Heering, J., and Klint, P., "Towards Monolingual Programming Environments", *ACM Transactions on Programming Languages and Systems*, 7-2, (April 1985), pp. 183-213.

[HerLis82]
Herlihy, M. and Liskov, B., "A Value Transmission Method for Abstract Data Types", *ACM Transactions on Programming Languages and Systems*, 4-1, (Oct. 1982), pp. 527-551.

[He86]
Heuring, V. P., "The Automatic Generation of Fast Lexical Analysers", *Software Practice and Experience*, 16-9, (Sept. 1986), pp. 801-808.

[Lamb87]
Lamb, David A., "IDL: Sharing Intermediate Representations", *ACM Transactions on Programming Languages and Systems*, 9-3, (July 1987), pp. 297-318.

[Lin84]
Linton, Mark A., "Implementing Relational Views of Programs", *SigPlan Notices*, 19-5, (May 1984), pp. 132-140.

[Moss86]
MossenBock, H., "Alex - A Simple and Efficient Scanner Generator", *SigPlan Notices*, 21-12, (December 1986), pp. 139-148.

[Opp80]
Oppen, Derek C., "Prettyprinting", *ACM Transactions on Programming Languages and Systems*, 2-4, (Oct. 1980), pp. 465-483.

[Pag81]
Pagan, F. G., *Formal Specification of Programming Languages*, Prentice-Hall, (1981).

[PeSi83]
Peterson J. and Silberschatz A., *Operating System Concepts*, Addison-Wesley, (1983).

[PolSte80]
Pollack, S. V. and Sterling, T. D., *A Guide to Structured Programming and PL/I*, 3rd edition, Holt, Rinehart and Winston, (1980).

[PurBro81]
Purdom, P. W., Brown, C. A., "Parsing Extended LR(k) Grammars", *Acta Informatica*, 15-2, (1981), pp. 115-127.

[Rub83]
Rubin, Lisa F., "Syntax-Directed Pretty Printing", *IEEE Transactions on Software Engineering*, 9-2 (March 1983), pp. 119-127.

[Sun85]
External Data Representation Protocol Specification, Release 2.0, Sun Microsystems Inc., Ca., (1985).

[TeiReps81]
Teitelbaum, T., Reps, T., "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", *Communications of the ACM*, 24-9, (Sept. 1981), pp.563-573.

[Wir85]
Wirth, N., *Programming in Modula-2*, Third, Corrected Edition, Springer-Verlag, (1985).

[Wood86]
Woodman, M., "Formatted Syntaxes and Modula-2", *Software - Practice and Experience*, 16-7, (July 1986), pp. 605-626.

INDEX