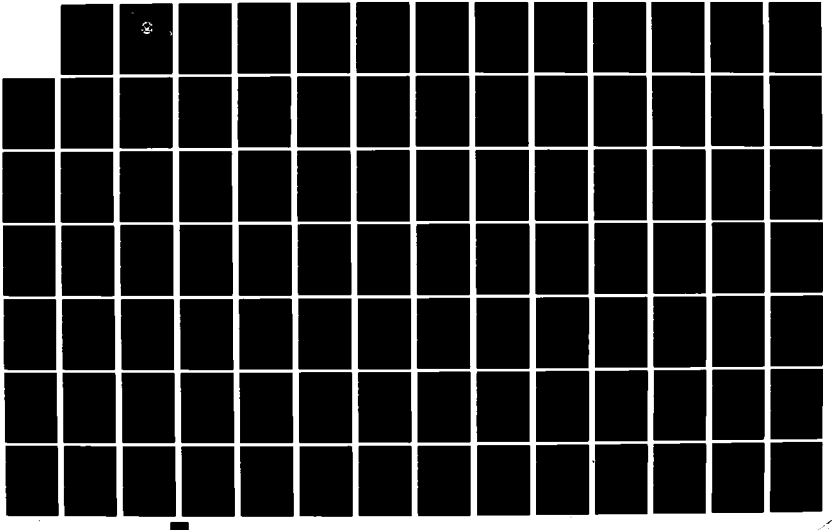
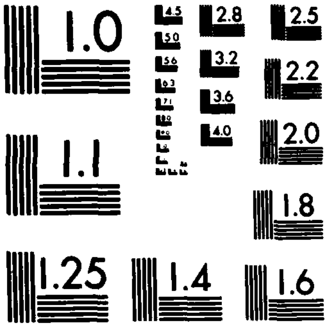
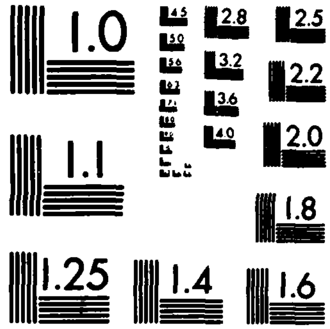
AD-A120 399 FUNCTIONAL PASCAL: AN INTERIM SOLUTION TO A CHANGING COURSE IN PROGRAMMING LANGUAGE DEVELOPMENT(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA O D BORCHELLER ET AL. 1/2

UNCLASSIFIED JUN 82 F/G 9/2 NL

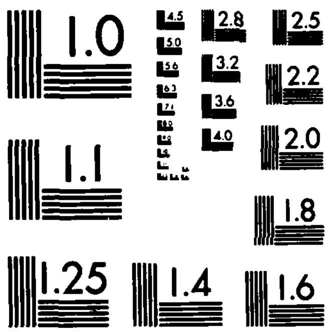




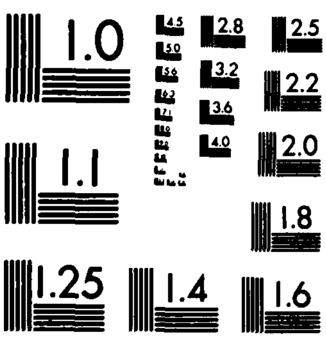
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



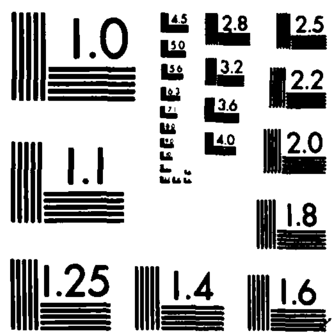
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



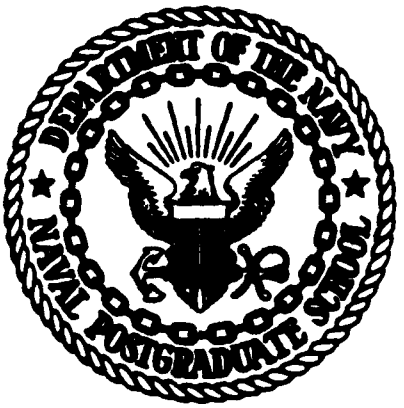
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(2)

NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

FUNCTIONAL PASCAL: AN INTERIM SOLUTION TO A CHANGING COURSE IN PROGRAMMING LANGUAGE DEVELOPMENT

by

Otis Dennis Borcheller

and

Ron Scott Ross

June 1982

Thesis Advisor: Bruce J. MacLennan

Approved for public release; distribution unlimited

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A120 395 / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Functional Pascal: An Interim Solution to a Changing Course in Programming Language Development	5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June 1982
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Otis Dennis Borcheller and Ron Scott Ross	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940	12. REPORT DATE June 1982
	13. NUMBER OF PAGES 164
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

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

18. SUPPLEMENTARY NOTES

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

Functional Programming, Applicative Programming, Functional PASCAL, List Processing, PASCAL, Programming Languages, Very High Level Languages.

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

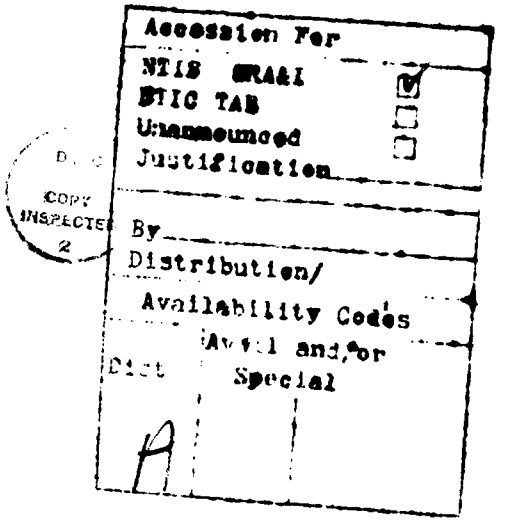
The theory of pure functional programming is applied to the standard conventional programming language PASCAL, thereby offering a unique and innovative language for problem-solving. A basic set of primitive functions and functional forms, as outlined in the Backus FP System, provides a model for the development of a practical functional programming system. This system is activated by accessing a detailed and comprehensive system library module directly from a PASCAL program, thereby enabling the user to operate

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

in either a functional or a conventional mode. The ability to perform functional programming within a conventional, high-level language, adds an increased degree of power and flexibility to the proposed system. The Functional PASCAL System provides the user with a new and distinctive methodology for writing computer programs and encourages individuals to experiment, in a practical environment, with functional programming techniques not otherwise available for general purpose use.



Approved for public release; distribution unlimited

Functional PASCAL: An Interim Solution to a Changing Course in Programming Language Development

by

Otis Dennis Borcheller
Captain, United States Army
B.S., United States Military Academy, 1971

and

Ron Scott Ross
Captain, United States Army
B.S., United States Military Academy, 1973

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June, 1982

Author: Otis Dennis Borcheller

Author: Ron Scott Ross

Approved by: Bruce MacLennan
Thesis Advisor

Douglas R. Smith
Second Reader

Chairman, Department of Computer Science

W. M. Woods
Dean of Information and Policy Sciences

ABSTRACT

The theory of pure functional programming is applied to the standard conventional programming language PASCAL, thereby offering a unique and innovative language for problem-solving. A basic set of primitive functions and functional forms, as outlined in the Backus FP System, provides a model for the development of a practical functional programming system. This system is activated by accessing a detailed and comprehensive system library module directly from a PASCAL program, thereby enabling the user to operate in either a functional or a conventional mode. The ability to perform functional programming within a conventional, high-level language, adds an increased degree of power and flexibility to the proposed system. The Functional PASCAL System provides the user with a new and distinctive methodology for writing computer programs and encourages individuals to experiment, in a practical environment, with functional programming techniques not otherwise available for general purpose use.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

The future of programming language development within the computer industry is indeed an area of great concern and strong debate. The events of history and the evolution of the science to its current state have created some difficult problems requiring systematic, bold and creative solutions. In order to fully understand the nature of the "computer crisis" of the 1980's, it is important to examine the course which charted the industry to its present position.

The cornerstone of computer science today is the conventional programming language and its associated von Neumann machine. The von Neumann model, which launched the modern era of computing machines over three decades ago, had as its basis the concept of word-at-a-time processing or the transfer of a single word between the CPU and storage area. While the initial development of the von Neumann computer was a profound technological breakthrough, serious deficiencies and limitations have been discovered in later research. John Backus has cited the most serious defect in this particular architecture as the "von Neumann bottleneck" resulting from the word-at-a-time programming concept [1]. The continuing reliance on the von Neumann computer has greatly influenced the direction of programming language development for the past thirty years. The conventional

programming language, with its foundation in the word-at-a-time methodology, has flourished since its inception and has served the user community extremely well. However, an interesting phenomenon is occuring within the conventional language environment. The total dependence upon von Neumann computers by both the business and academic communities has literally forced the industry into developing, for the most part, conventional-type programming languages. In an effort to make stronger and more sophisticated languages, it became necessary to increase their overall size and complexity. As a result, the state-of-the-art development of conventional languages is at present producing a much larger but not technologically or intellectually superior product.

Computer language technology has progressed through a multitude of stages from the original von Neumann style of programmi g languages to what appears to be a turning point in the future design and implementation. Standing at the crossroads in this continuing developmental process are many prominent voices urging movement along a variety of paths. The traditionalists remain comfortable with the von Neumann style of programming languages while the mavericks of the Backus philosophy insist that the conventional languages have been pushed to their limit and the many inherent weaknesses of these languages must be overcome by introducing the concept of a functional style of programming. The notion of functional programming, as

espoused by Backus, contrasts the inherent problems of the conventional languages, i.e. the von Neumann bottleneck and word-at-a-time mentality with an alternate methodology of thinking and reasoning on a broader, more conceptual level. The functional languages, along with the object-oriented languages such as Smalltalk from the Xerox Corporation and the natural languages from the artificial intelligence arena, offer a wide variety of alternatives to the von Neumann model.

It is readily apparent that the architecture of the original von Neumann machines has not only influenced the past development of languages, but also continues to stagnate the intellectual growth of the science, especially in bold new areas such as functional programming. The lack of suitable machines, in general, to process non-von Neumann languages has reinforced the development and refinement of conventional languages, thus, creating a vicious cycle which may be very difficult to overcome. In addition, the ability to gain the acceptance of the user community for a new style of programming will be at best very difficult and at worst virtually impossible. It is essential, therefore, that a detailed and comprehensive plan be established industry-wide to facilitate not only the future growth of non-von Neumann languages but also the development of non-von Neumann architectures to support these languages. The solution to the problem must be sought in two distinct areas; (1) the development of new hardware technologies which will be able

to process the non-von Neumann programming languages in an efficient and cost-effective manner and (2) the development of an intermediate programming language which will enable the users of conventional programming languages to slowly and systematically learn and accept a new style of programming and ultimately a more sophisticated and powerful methodology for problem solving.

B. SCOPE

The intent of the thesis is to explore a possible interim solution to the long range problem of implementing a non-von Neumann type of programming language and in addition, gain user community acceptance of this powerful, unconventional programming technique. The growing problem of programmer productivity and associated human factors considerations related to the programming process are central issues in the motivation for the thesis. It is assumed that the hardware issues previously mentioned will eventually find a solution with advancing technologies and, therefore, have not been included in the scope of study. The thesis proposes to show the potential value of integrating a non-von Neumann programming method into a current conventional language with the sole purpose of providing a smooth, systematic and orderly transition into the future generations of programming languages.

1. Statement of the Problem

There is currently no method for a programmer to use the powerful concepts of functional programming within a

conventional language environment as an interim solution to the development and implementation of advanced programming language techniques.

2. Hypothesis

The theory of pure functional programming can be applied to a standard conventional programming language. A subset of the total concept and problem-solving techniques of a functional programming language can be implemented within the structure of a conventional language, thus enabling a programmer to simulate a functional programming environment while remaining within the traditional bounds of a conventional language.

3. Dependent and Independent Variables

In order to fully describe the interrelationships between the different components involved in the research endeavor, it is necessary to identify the major dependent and independent variables. The non-von Neumann methodology selected for integration into a conventional environment is the concept of functional or applicative programming as described by Backus [1]. The functions developed for integration will be termed the independent components or variables. The high-level conventional language chosen for the research is the PASCAL programming language. It will serve as the dependent component or variable inasmuch as its overall capabilities will be significantly altered by the implementation of the independent functions which will be integrated into its structure.

4. Intended Contribution

The contribution of the research effort is focused primarily on the issue of programmer productivity and in particular, the necessity for providing new and innovative tools for the development of computer programs. It is intended to be a small step towards interfacing with the programming language of the future which by its very nature will be simple, yet powerfully extensible. The conventional languages will be embedded in the majority of applications in the near term as it is very difficult to change traditional programming habits. A successful test of the above hypothesis will convince the reader that there is indeed, from the users point of view, an interim solution to bridging the gap between the current conventional programming languages and the powerful unconventional languages of the future. It will reinforce the concept that a programmer can simulate the techniques and problem-solving methods of a functional or applicative type language while using the traditional tools to which he or she is accustomed. The proposed language or system, termed Functional PASCAL, will offer a glimpse into the future of programming language design. It is not intended to be a complete solution but will hopefully achieve two important objectives; (1) provide a vehicle for exploring the use of a functional programming language and (2) provide a planned methodology for achieving the ultimate goal of developing a simple, yet powerful programming language for the user.

As the cost of hardware continues to decline dramatically within the industry, the explosion in microtechnology is driving the development and production of faster, less expensive and more efficient machines. In contrast, the problems facing mankind are growing progressively larger and more complex along with the software used to solve those problems. It is essential, therefore, that appropriate tools be developed to meet the challenge of a more complicated world. The largest portion of the computer budget in the future will, without a doubt, be targeted for the software arena. It is imperative that the methods for building, maintaining and understanding computer programs be addressed in a bold and innovative manner. It is with this thought in mind that the following research is accomplished.

C. OVERVIEW

A brief introduction has been provided in Chapter I outlining the general background of the problem and the scope of the study. Chapter II focuses on a detailed review of the literature and addresses some of the basic concepts and definitions within the realm of functional programming. Alternative solutions for integrating the concepts of functional programming into a conventional language environment are explored in Chapter III. The proposed design and detailed specifications for the final development of the Functional PASCAL System are provided in Chapter IV. Chapter V contains the implementation procedures for the system and

the initial test results in an operational environment. The study is discussed in its entirety in Chapter VI, rendering pertinent comments about the findings of the research, a final summation of the results and the conclusions of the researchers. Recommendations for further research are also addressed along with related topics of interest in the area of functional programming. A complete documentation package is provided in Appendix A (Source Code) and Appendix B (User's Manual).

II. REVIEW OF THE LITERATURE

A. GENERAL

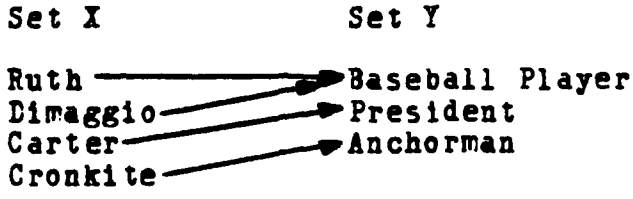
The literature does not, at present, specifically address the issue of integrating a functional or applicative language into a conventional environment. However, a general overview of the theory, concepts and structures of functional programming is deemed essential in order to fully understand the purpose and ultimate benefit of the proposed system. In an attempt to prove the validity of implementing a non-von Neumann type programming language in a conventional environment, it is first necessary to explore not only current ideas and methodologies, but also the basic terminology that comprises the new technique.

As was mentioned earlier, the declining cost of hardware is allowing the designers of programming languages to concentrate more on building simple and extensible languages and less on the efficiency of implementation. The escalating cost of software, on the other hand, reinforces the theory that more emphasis needs to be placed upon the nature of the programming language itself. Functional programming enables a user to experience an even higher-level programming capability than is provided with traditional conventional languages. Programs are easier to construct, easier to maintain and easier to understand [2]. The following review of the literature and explanation of basic concepts will

provide the reader with a better understanding of the world of functional or applicative programming. Individuals with previous knowledge of functional programming may wish to skip Chapter II and continue with Chapter III, the alternative solutions to the problem.

B. BASIC DEFINITIONS AND CONCEPTS

Functional programming is an unconventional style of problem-solving which traces its history to the simple concept of the mathematical function. The basic concept of a function is a rule of correspondence which maps members of set X, for example, into a unique member of set Y. The members of set X comprise the domain and the members of set Y, the range [2]. The members of the two sets are objects and the resulting operation of the function is a mapping of each object from the domain into a unique object in the range.



The function is considered the operator and the object, the operand. In the example, the function OCCUPATION is applied to the object Ruth and the result is another object, Baseball Player. The object Ruth can also be thought of as the argument passed to the function, OCCUPATION. It is essential to reiterate that only unique mappings can occur from Set X to Set Y, i.e., members of Set X cannot map to

two distinct objects in Set Y. It should also be noted that there may be bounds placed upon the allowable members of the domain and range. For example, the function SQUAREROOT has as its domain and range the set of non-negative real numbers. The function SQUARE, however, has as its domain the set of real numbers and as its range, the more limited set of non-negative reals. To generalize the notion, it can be stated that a function takes as its argument, an object, and produces another object.

f (object) ====> (object)

A function can be thought of as a program which takes input in the form of an argument and produces some output in the form of a result. The basic structure of expressions in a functional language consists solely of operators applied to operands [2]. The operands are the actual values or arguments that are passed to the functions while the functions serve as the operators which take the values and operate on them producing a result. This result, which is also a value, can be used as an argument to another function if desired.

In order to take advantage of the powerful concept of functions in building a program, it is essential that a user be allowed to construct new functions from old ones. The building of these new functions from ones previously defined is called functional composition and provides the capability of implementing a hierarchical structure within the program. In its simplest form, a functional program consists of one

basic primitive function. At the other extreme, the program consists of a hierarchy of functions built from other functions resulting in the view of the entire program as one large function. There are two distinct schools of thought as to the implementation of the basic primitive functions within the language and these ideas will be discussed later.

The differences between a conventional programming language of the von Neumann mold and the functional or applicative languages described by John Backus and Peter Henderson are numerous. The basic difference is that "imperative languages" such as PASCAL, COBOL and FORTRAN compute by effect rather than by value. Computing by effect involves the incremental changes to variables by a continuous series of assignment statements [2]. In contrast, functional programs compute by value or, in other words, a unique value of a particular function is determined by its arguments. The imperative languages, therefore, are more assignment-oriented and variable-oriented in nature than applicative languages. An applicative language, such as functional programming, is more function-oriented and value-oriented with additional emphasis on the concept of recursion. In addition, applicative languages are usually organized around functional applications. Since a function operates only on its arguments to produce a result or value and there is no ultimate assignment to a variable, the need for variables is eliminated. The actual arguments are not named, either, which also eliminates the need for variables

as they have been used in the past. A functional program does not incorporate any data, per se, but operates instead on whole conceptual units. The basic power of the applicative type of system is the ability to build functions from functions without the use of variables, thus creating a simple, yet elegant hierarchical structure for a program. Programs written in functional or applicative languages are, in many cases, an order of magnitude shorter than equivalent conventional language programs [2]. Although the theory of functional programming has been studied extensively and many different models proposed, the review will concentrate on the functional programming system proposed by John Backus [1]. References to other systems and contrasting methodologies will be made in order to give the reader a balanced perspective of the research and development that has been completed to date. It should be emphasized, however, that the development of a subset of a functional programming language for integration into a conventional environment as proposed in subsequent chapters of this paper, is accomplished using the basic framework of the Backus system.

C. A FUNCTIONAL PROGRAMMING SYSTEM

A functional programming system, according to Backus, is built from five component parts; (1) a set of objects, (2) a set of functions, (3) operations or applications, (4) a set

of functional forms and (5) a set of definitions [1]. Each of these component parts is explained in greater detail below.

1. Objects

An object, as mentioned earlier, is an argument that is passed to a function which it operates on. It can either be an atom, in its most basic sense, or a sequence where the sequence itself is a set of one or more objects. An object that is neither an atom nor a sequence is said to be undefined and is denoted by a special symbol, an inverted T (called "bottom") [1]. The allowable symbols for an atom within the system are digits or nonnull strings of alphabetic letters. Cetain special symbols may also be used if they are not implemented as a feature within the language itself. Henderson classifies atoms as either symbolic or numeric and uses the notion of an "S-expression" to illustrate how atoms are combined to form simple and complex sequences [2]. The most basic form of an "S-expression", according to Henderson, is a single atom, while a more sophisticated "S-expression" might contain a list of atoms or sequences of atoms. A special symbol is used to denote the empty sequence and is the only object in the functional system that can serve as either an atom or a sequence. The actual representation or specific symbol used depends on the particular designer of the system but the most common one is the symbol for the null set (Ø).

Some Elementary Objects in a Functional Programming System:

ATOMS: B C X Y BALL BAT 1 5 8.4

SEQUENCES: <A B C> <X Y <X Y Z>> <<E> <F G> <H I J>>

<<CAT BALL 2> <<1.2> <6.5>> 3 DISH>

Again, it must be emphasized that everything in a functional system is viewed in terms of functions or operators applied to objects or operands. The single atoms and combination of atoms into sequences illustrated above have the common bond in that they all represent objects.

2. Functions and Applications

Now that the concept of an object has been defined, a logical follow-on component is the function that provides the actual operation on those objects. A function has the responsibility of mapping objects into objects. The act of operating on an object by a function is termed an application. It derives its name from the fact that the function, or operator, is applied to a specific object, or operand, to obtain a result [1]. Most functional programming systems provide different types of functions that can be implemented by the user. In the functional system designed by Backus, a function can be in one of three catagories; (1) primitive, (2) user-defined or (3) functional form. The primitive functions are those that are provided by the system whereas the user-defined functions are developed for specific applications by the user through the use of one or more of the primitives. The Backus philosophy encourages the use of a powerful and diversified set of primitive functions

which provides the programmer with a significant amount of latitude in designing a functional program. In contrast, the system proposed by Henderson limits the set of primitive functions to elementary selector and constructor functions such as FIRST, REST and CONS (CAR, CDR and CONS in LISP notation). It is the user's responsibility to define new functions in terms of the primitive ones provided by the system, which places an added burden on the programmer.

Examples of Applications Using Primitive Functions:

FIRST : <X Y Z> ===> X

REST : <<BALL> X YZ <BAT>> ===> <X YZ <BAT>>

CONS : <<X Y> <<X Z> <BALL BAT>>>

===> <<X Y> <X Z> <BALL BAT>>

LENGTH : <21 BAT <X Y> B> ===> 4

REVERSE : <1 2 3 4 5> ===> <5 4 3 2 1>

In addition to the elementary selector and constructor primitives, most functional systems provide, either as a user-defined function or as part of the system-defined primitives, arithmetic and predicate functions. The need for the former is obvious while the latter becomes an essential feature of the language in order to process lists of varying structure [2]. A predicate function is one that returns a boolean value when it is applied to its argument. Having the predicate capability enables a programmer to test a structure to determine if it is an atom or a list or whether or not a particular structure is equal to another structure.

Examples of Predicate and Arithmetic Functions:

SUM : <1 10 4 6> ===> 21

ATOM : <BALL BAT> ===> False

EQ : <RUN RUN> ===> True

3. Functional Forms

The true power and sophistication of a functional programming system is achieved through the mechanisms of functional definitions and functional forms. The use of functional forms is unique to the system implemented by Backus while user-defined definitions are usually a standard feature of most functional programming systems. The functional form is a mechanism used to combine existing functions or objects into new functions [1]. The arguments of a functional form can be functions or objects. The methodology of building functional forms may be familiar to the reader as "combining forms" and is explained in greater detail by Curry and Feys in their work on combinatory logic [3].

Examples of Functional Forms:

COMPOSITION: Symbol ===> o

(Assuming two functions f and g)

Functional form: f o g

Application to object x: (f o g) : x

Represents: f : (g : x)

APPLY TO ALL: Symbol ===> a

(Assuming one function f)

Functional form: af

Application to object X

where X is a sequence: af:<X1 X2 X3>

Represents: <f:X1 f:X2 f:X3>

4. Functional Definitions

The user-defined function rounds out the last component of a functional programming system. It gives the programmer an opportunity to define new functions using the principles of combining forms and actually names the function for future use. Although most functional systems allow the user to specify new functions, the actual implementation may vary from system to system. The basic concept, however, remains the same. There is a relationship established where the left-hand side of the equation consists of the symbol selected for the newly defined function. The right-hand side of the equation, then, is a set of primitive functions or a functional form. When the symbol for the new function is invoked, it has the net effect of substituting the pre-defined functions or functionals for the user-defined symbol, thus creating the new function. This method for defining functions provides the programmer with a powerful capability of extending the language and also offers an added dimension of flexibility to the system.

Examples of User-Defined Functions:

```
(Define a function that returns the second
 element of a list)

DEF SECOND <X> = <FIRST <REST <X>>>

(Application of the new function)

SECOND : <3 6 9>  ===>  6
```

The evaluation of the new function is accomplished by substituting the previously defined primitives, which in turn requires two more function applications. The function REST applied to the object X, which is a list containing three atoms, returns an object. This object, a list containing the two atoms 6 and 9, is passed to the function FIRST which returns another object, the atom 6. Although a trivial example, the concept of creating new and more powerful functions from old ones is indeed a desirable property for any programming language and is an essential feature of functional systems.

I. ADDITIONAL CONCEPTS AND METHODOLOGIES

It should be noted that the references to objects in the functional programming system are actually values and should not be confused with the purely mathematical definitions as discussed by MacLennan [4]. The applicative programming system remains value-oriented in nature by definition.

With all of the advantages of a functional programming system come several disadvantages as well. One of the primary limitations is the inability to add functional forms to the system when needed. The standard functional system, then, is rigidly fixed; its capability is limited by the original set of primitive functions and functional forms that establish the framework of the language. The only flexibility in the functional system is the programmer's option of defining a function by name, using the functional forms provided within the system. In an attempt to overcome

this restriction and recover the simplicity and elegance attributed to applicative systems, the notion of a formal functional programming system (FFP system) is introduced [1]. The FFP system provides the programmer with the capability of defining new functional forms and ultimately increases the overall flexibility of the system.

Chiarini extended the functional programming concepts of Backus by implementing an enhanced version of an applicative system [5]. He not only standardized the symbols used in a functional programming system so that programs could be implemented on ASCII terminals, but also developed additional functional forms that complemented the original ones proposed by Backus. The functional forms proposed by Chiarini are examined in greater detail in Chapters 3 and 4.

Another limitation of a functional or applicative system is its lack of history sensitivity [1]. The original concept of eliminating the notion of states from programming was motivated by the overwhelming complexity that resulted from programs having to keep track of thousands of variables, representing the state of a machine at a particular point in time [4]. In contrast to the large number of state changes required by an imperative language due to the constant alteration of variables by assignment statements, the concept of functional programming presents a completely different approach. A reduction in the number of state transitions is a highly desirable property, but the complete absence of states altogether in the functional system

proposed by Backus imposes some severe restrictions. The need for compromise resulted in a proposal for an Applicative State Transistion System (AST System) [1]. In this system, a new state is computed by the application of a particular function to its argument or arguments which produces an output or result. The output is, in fact, the new state. A critical feature of the AST System is that no state changes occur during a computation; only upon completion of the functional application. The significant reduction in state transitions greatly simplifies the underlying computation and brings together the best of both programming methodologies. The actual components of an AST System consist of a formal functional programming system as previously described, a set of definitions that determine the state of the program and a set of transition rules that allow functional applications to occur and affect changes to the original state [1].

Any dissertation on functional programming must include a discussion of recursion. Generally speaking, a functional system consists of functions operating on objects. These objects are represented in a list structure and therefore, resulting operations imply changes to that list. Two cases must be considered when applying recursive functions to a list; (1) the situation when the list contains one or more elements and (2) the situation when the list is empty. By implementing a recursive function and applying it repeatedly to an object (list), the recursion is guaranteed to

terminate. The list decreases in size by one item with each recursive call until the list is empty and the null condition is reached [2]. The concept of recursion is extremely useful when implementing new functions which operate on lists and is a powerful tool that is a welcome feature in a functional system.

A Recursive Function:

```
LENGTH <X> =
     IF EQ : <X,nil> THEN
          0
     ELSE
          LENGTH : <REST<X>> + 1
     END IF
```

This cursory review and explanation of terminology is intended to give the reader a broad overview of the concept of functional programming. The subsequent chapters attempt to use a subset of these powerful programming techniques in a controlled manner within the framework of a conventional programming language.

III. ALTERNATIVE SOLUTIONS TO THE PROBLEM

A. GENERAL

In examining the various alternatives to implementing functional programming techniques within a conventional programming language environment, it is important to remember the most critical factor and overall goal of the system design: ease of use for the programmer. It is this basic premise that is so often ignored in many language designs. The net result is the development of highly sophisticated programming languages which, for the average individual, are extremely difficult to understand and use effectively. To maintain the proper perspective, designers must be aware that the ultimate purpose of computers and the languages which provide their "intelligence", is to accomplish real-world tasks. As problem solving becomes more complex and the availability of computers to the average person becomes more widespread, the task of providing simple and powerful programming languages to a large and diversified user community is, indeed, a challenging one.

B. PROGRAMMING LANGUAGE STRUCTURE

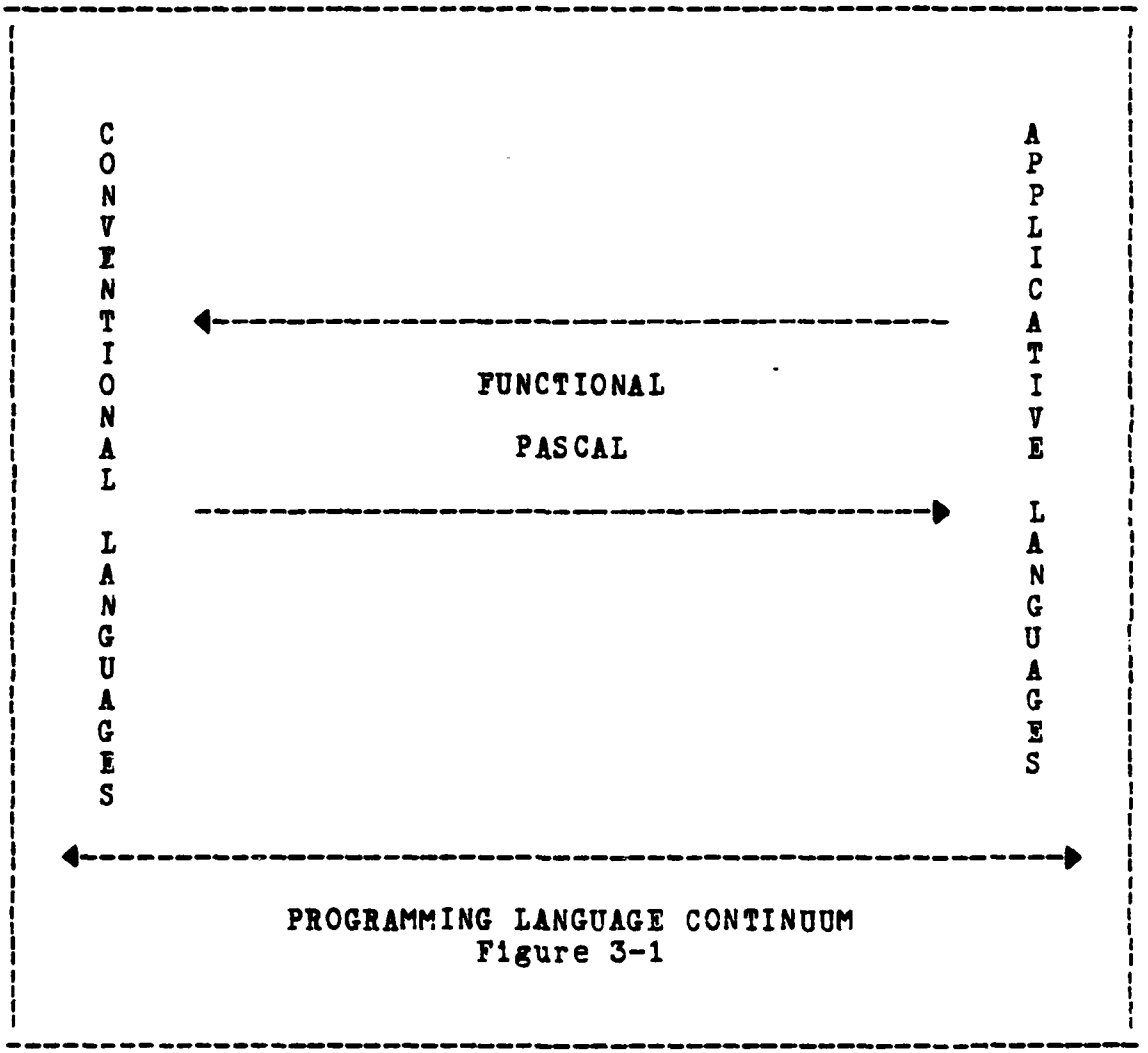
1. The Programming Language Continuum

The current technology of programming language development can be viewed as a continuum with two distinct endpoints. One extreme represents the standard imperative languages, while the other extreme represents applicative

languages. The number of existing languages within the former class are numerous while the number of operational languages within the latter are few and far between. The design of the Functional PASCAL System attempts to provide the user with the capability of operating at any point on this "programming language continuum" as illustrated in Figure 3-1. The left side of the continuum represents the use of high-level conventional programming languages for a particular application. The right side represents the use of applicative programming techniques to accomplish a programming task. The purpose of the Functional PASCAL System is to enable an individual to operate at a point in between the two extremes. Ultimately, the system will allow the user to simulate the techniques of functional programming to any degree desired or remain in the familiar conventional environment. The degree to which functional versus conventional programming techniques are used depends, to a large extent, on the type of calculation required and the motivation of the user to experiment with new methodologies.

2. Programming Language Components

Programming languages, in general, consist of two basic components; (1) a framework and (2) a series of changeable parts [1]. The framework dictates the basic rules of the system and describes the fixed features of the language. The standard language constructs, i.e. the IF, CASE, WHILE, REPEAT and FOR statements are examples of

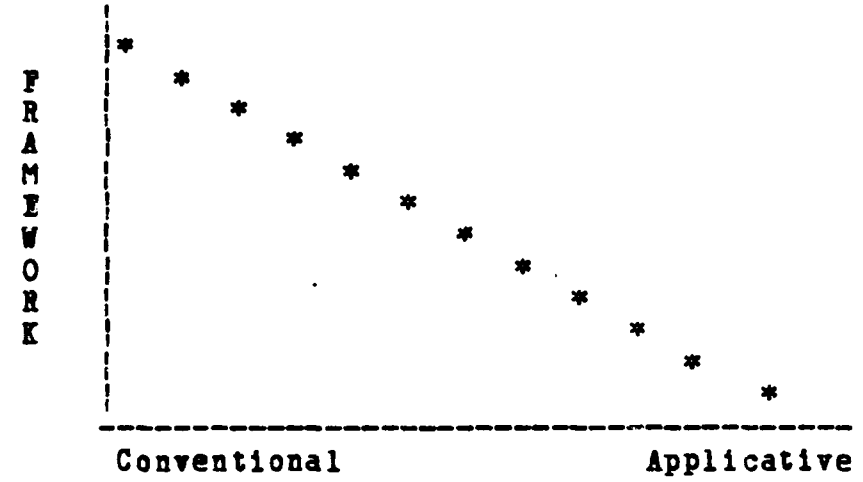


PROGRAMMING LANGUAGE CONTINUUM
Figure 3-1

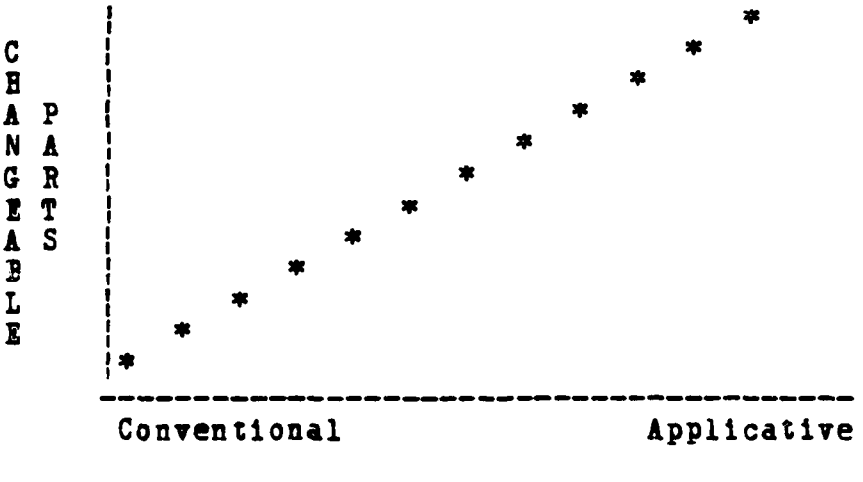
portions of the PASCAL language framework. Changeable parts, on the other hand, are independent components such as user-defined procedures or library functions which exist within the framework and provide a degree of flexibility to perform certain operations.

Some of the current state-of-the-art conventional programming languages seem to consist of large, bulky frameworks with moderate to small numbers of changeable parts; this results in the inflexibility of von Neumann style word-at-time programming. The underlying principle of the Backus philosophy envisions reduced frameworks and a marked expansion in the number of changeable parts which will provide the programmer with increased control and flexibility over the programs being created. Figures 3-2 and 3-3 illustrate the relationship between "frameworks" and "changeable parts" as viewed across the programming language continuum. It is hypothesized that there is a noticeable decrease in the size of the language framework as the continuum is traversed from conventional languages to applicative languages. In contrast, the number of changeable parts seems to show a complementary increase over the same continuum, reinforcing the hypothesis that frameworks can be reduced as the repertoire of changeable parts grows.

In attempting to provide a long-term solution to the problem of large, unwieldy frameworks and insufficient changeable parts within the conventional programming languages, a purely applicative language could be developed



LANGUAGE-FRAMEWORK RELATIONSHIP
Figure 3-2

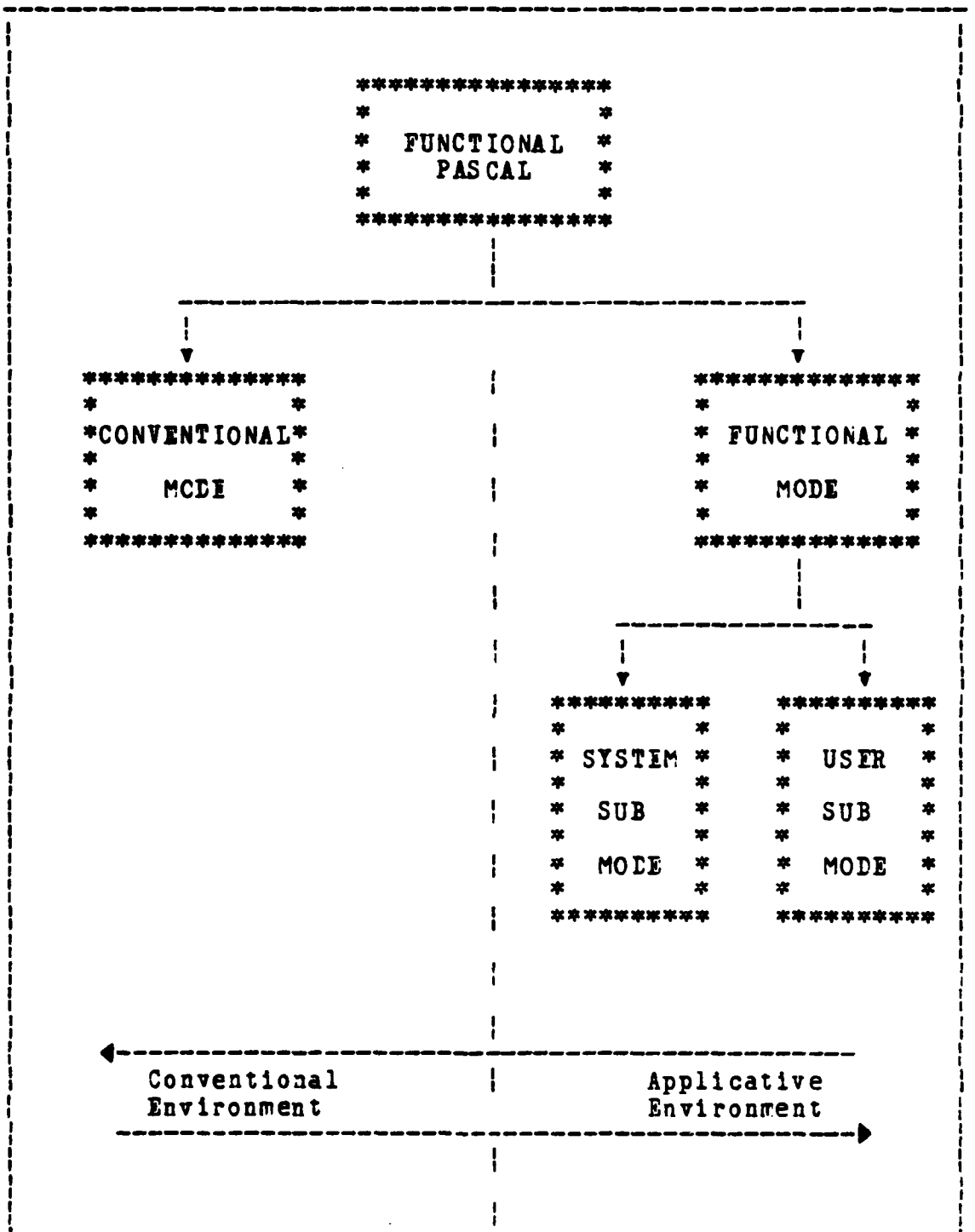


LANGUAGE-CHANGEABLE PARTS RELATIONSHIP
Figure 3-3

along the lines of the Hendersen model [2]. However, in the short-term, the critical factor that must be considered is user reluctance to use a functional language in total for an entire programming application. It seems that human beings not only think in a sequential manner, but are also creatures of habit and occasionally tend to resist change. With these facts in mind, it is essential that the proposed system allow the user to decide when functional programming will be used (if at all) and to what degree. The user must be given the flexibility to move across the "continuum" at will in order to achieve the degree of functional programming that is comfortable and useful for the particular application. When the desired results are achieved, the programmer can return to the sanctity of the conventional environment and continue the application.

C. FUNCTIONAL PASCAL: THE BASIC CONCEPTS

The methodology for achieving the transition from conventional to functional programming techniques is three-fold: (1) increase the number of changeable parts within the language by providing a rich and powerful set of primitive functions as presented by Backus, (2) reduce the dependence on the conventional language constructs by allowing the programmer to implement user-defined functions built from the basic set of primitives, and (3) provide the programmer with a clear and concise choice as to which environment is best suited for the particular application [1].



FUNCTIONAL PASCAL SYSTEM
LOGICAL SCHEMA
Figure 3-4

The overall logical schema of the Functional PASCAL System is illustrated in Figure 3-4. The system consists of two basic logical modes of operation: (1) conventional and (2) functional. The user is provided the opportunity to select either mode and subsequently operate in that mode for all or part of the application. If the user decides to operate in the functional mode, there are two logical sub-modes from which to choose, again, dictated by the application and type of calculations required. The "system" sub-mode allows the programmer to access the system-defined primitive functions which are resident in a library. The "user" sub-mode allows the programmer to define new functions within the program by using the standard system-defined primitives from the library. The logical modes and sub-modes are discussed in detail below.

1. Conventional Mode

The conventional mode of operation allows the user to operate purely in a high-level, conventional programming language. In the prototype Functional PASCAL System, the high-level language used as the foundation of the system is the University of California (Berkeley) PASCAL, which is designated the "host" language. UC Berkeley PASCAL runs under the UNIX operating system from Bell Laboratories. The hardaware selected for system development and implementation is the VAX Model 11/780 from the Digital Equipment Corporation (DEC). The basic concept of the system, however, is flexible enough to be designed around any of the

widely-used, high-level programming languages and run on compatible hardware configurations. This would enable similar systems such as Functional COBOL and Functional FORTRAN to be developed in the future.

2. Functional Mode

The heart of the Functional PASCAL System is the notion of the primitive function as described by Backus in his research and writings on functional programming [1]. These "primitives" form the building blocks of the functional system and play an important part in the proposed hybrid system. In order to perform functional programming in the PASCAL environment, the primitive functions are programmed in the host language and reside in a system library. The programmer is able to use any of the "primitives" to simulate a functional programming operation within the PASCAL program. The basic primitive functions provided in the Functional PASCAL system are:

(1) selector	(9) length	(17) distribute from left
(2) tail	(10) add	(18) distribute from right
(3) identity	(11) subtract	(19) append left
(4) atom	(12) multiply	(20) append right
(5) equals	(13) divide	(21) right selector
(6) null	(14) and	(22) right tail
(7) reverse	(15) or	(23) rotate left
(8) transpose	(16) not	(24) rotate right

These "primitives" operate on list structures which serve as the arguments to the functions. The result of this "function application" is a newly-formed list structure.

In addition to this comprehensive set of primitive functions, a limited number of functional forms are included in the system library. The functional forms are slightly

more complex entities than the basic primitive functions and can best be described as expressions with functional parts. The functional forms are different from the primitive functions inasmuch as the functions involved in the expression have an interdependent relationship. The functions, or dependent elements, serve as parameters to an independent element of the expression. The basic functional forms (functionals) provided in the Functional PASCAL System are:

(1) insert
(2) apply to all

The design and implementation of the primitive functions and functional forms are discussed in further detail in Chapters 4 and 5.

3. System Sub-Mode

The "system" sub-mode represents the portion of the logical schema which allows the programmer to access the primitive functions and functional forms in the system library and perform the basic operations of functional programming. The degree of flexibility in this logical sub-mode is limited to those operations which can be accomplished by the call of a single library function. The primitive functions can provide the user with 24 potential operations at the most basic level of functional programming. The functional forms offer an added dimension and increased capability for the "system" sub-mode. Individual documentation sheets are provided for each primitive function and functional form in Chapter IV.

4. User Sub-Mode

The "user" sub-mode represents the portion of the logical schema which allows the programmer to create new, user-defined functions from the "primitives" and "functionals" resident in the system library. Having the ability to develop new functions from existing ones using the technique of combining forms is a critical element in the potential power of functional programming. Only one additional functional form is required to perform the task of function composition. The process of composition can be accomplished using the existing framework of the host language and can be implemented very simply by defining a new function in the host language using the existing primitive functions and functional forms from the system library. The extent of the capability of the user-defined functions is limited only by the imagination of the programmer.

IV. FUNCTIONAL PASCAL: A PROPOSED DESIGN

The design of the Functional PASCAL System involves the augmentation of the basic, high-level, host language with a series of functions, both system and user-defined. Due to the magnitude and scope of the Backus proposal, the total system development effort is divided into two phases. Phase I encompasses the design and implementation of an interactive input function, an output function, 13 selected primitive functions, 2 functional forms and the capability to produce user-defined functions. Phase II is the design and implementation of the remaining 11 primitive functions and the more complex functional forms outlined in the Backus functional programming system. The scope of the development portion of this thesis, in general, and Chapter IV specifically, is limited to Phase I.

A. METHODOLOGY

The design of the Functional PASCAL System follows the top-down, structured approach using the logical schema shown in Figure 3-4 as its foundation. In addition, the principle of "information hiding" is enforced by specifying the exact interfaces between modules and nothing more. For example, the generalized input function always returns the location (a pointer) of the newly-created data structure. This location serves as an input parameter to the system (primitives and functionals) and user-defined functions. The

system and user-defined functions do not have any knowledge of the internal workings of the input and output functions, and vice versa. Thus, a "black box" approach is obtained. The development effort concentrates on the Functional Mode of the logical schema, as the PASCAL host language provides the vehicle for performing conventional programming tasks.

The functional portion of the proposed system necessitates the design of three major components: (1) an interactive input function, (2) a system library containing the primitive functions and functional forms outlined in Chapter III and (3) an output function.

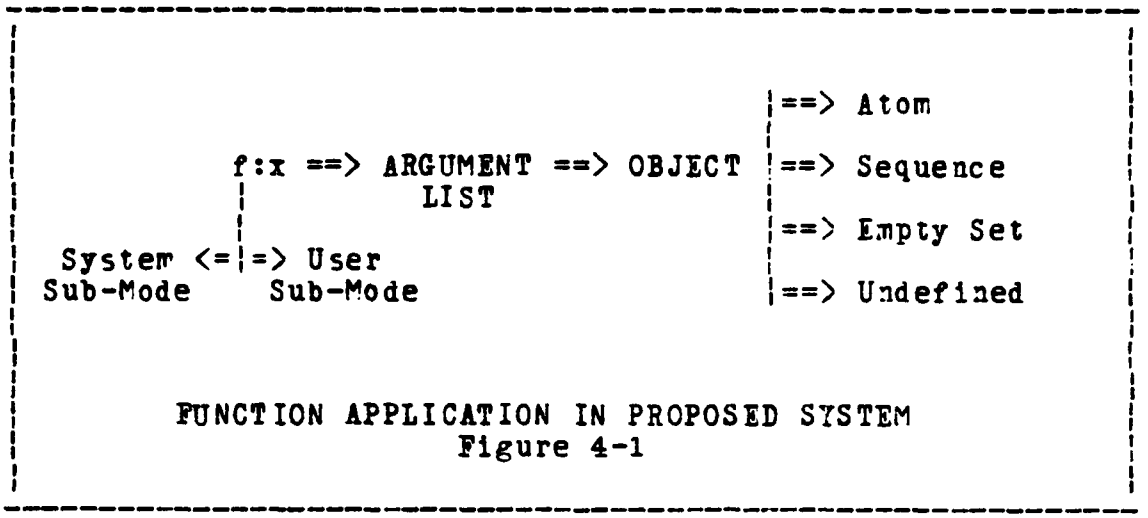
B. FUNCTIONAL MODE DESIGN

As discussed in Chapter II, the basic format for a functional programming operation is represented by the notation:

f : x (f applied to x)

where f represents a primitive function, functional form or user-defined function and x represents an argument list or "object" which can be an atom, a sequence or undefined (See Figure 4-1).

The Functional PASCAL System departs from the traditional Backus definition of "object" in the following manner. The functional programming system of Backus states that an "object" can be an atom, a sequence or an undefined entity and that the elements within a sequence can, in themselves, be "objects". The proposed Functional PASCAL System, on the other hand, defines "object" to be the entire



FUNCTION APPLICATION IN PROPOSED SYSTEM
Figure 4-1

argument list that the system and user-defined functions operate on. The component parts of this "object" are termed "elements" and can be either atoms or sequences. For purposes of this thesis, the terms "argument list" and "object" are synonomous.

In examining the functional portion of the Functional PASCAL System, the following generalized algorithm is provided as the first iteration in a "stepwise refinement" design process.

ALGORITHM FUNCTIONAL-MODE-DESIGN

- Input argument list (object)
- Create a data representation for the argument list (object) and return the location of the newly-created data structure
- Using the argument list, perform the operation specified by the function and return the location of the data structure as modified by the function
- Output the results of the functional operation
- Continue the application in the conventional or functional mode

END FUNCTIONAL-MODE-DESIGN

The procedures within the algorithm perform four very basic operations: (1) obtain the input data from the user, (2) place the data into an appropriate representation, (3) manipulate the data and (4) output the data. Each of these procedures is explained in detail in Sections 1-4 below.

1. Interactive Input Function

The purpose of the input function (REALLIST) is to interactively read a user-defined argument list from a CRT, create a doubly linked list data structure to represent the argument list and return a pointer to the data structure. The user enters the argument list from the terminal and it is placed in a temporary buffer. The exact format for entering the string into the system is outlined in detail in Appendix B, The User's Manual. After the argument list is input into the buffer, the parsing process begins in order to determine if the object is (1) an atom, (2) a sequence, (3) the empty set (a special form of a sequence) or (4) an undefined entity. If the object is determined to be a sequence, a procedure (GENLIST) is invoked which recursively creates a doubly linked list data structure for each sequence in the argument list. The following generalized algorithm is provided as the first iteration in the design of the interactive input function. The resultant source code is listed in Appendix A.

ALGORITHM INTERACTIVE-INPUT

- Input argument list from user

- Determine status of input string, i.e. atom, sequence, empty set or undefined entity

IF (input string is an atom) OR (input string is the empty set) THEN

    - Create an appropriate data representation

ELSE IF (input string is a sequence) THEN

    - Recursively generate an appropriate data representation for each sequence in the argument list

ELSE (input string is undefined)

    - Generate an error message

- Fill in administrative information and data for each record generated

- Return the location of the data structure for future access by system and user-defined functions

END INTERACTIVE-INPUT

2. Data Structure Creation

The data structure chosen to represent the input argument list in the Functional PASCAL System is the doubly linked list. The linked list data structure is dynamic in nature and allows the user to maintain a list which may be altered at random by the addition or deletion of components [6]. The procedure GENLIST, as part of the function REALLIST, creates a doubly linked list structure for each of the sequences in the input string. The result is the production of a list of lists with each node being represented by a variant record with designated fields. The design of the record format provides the capability to

retain both administrative information and data related to the actual calculation. The structure and content of the record is determined by the value of two "tag" fields, NODETAG and DATATAG. These "tag" fields indicate whether the node represents data or a list, and if data, what type of data.

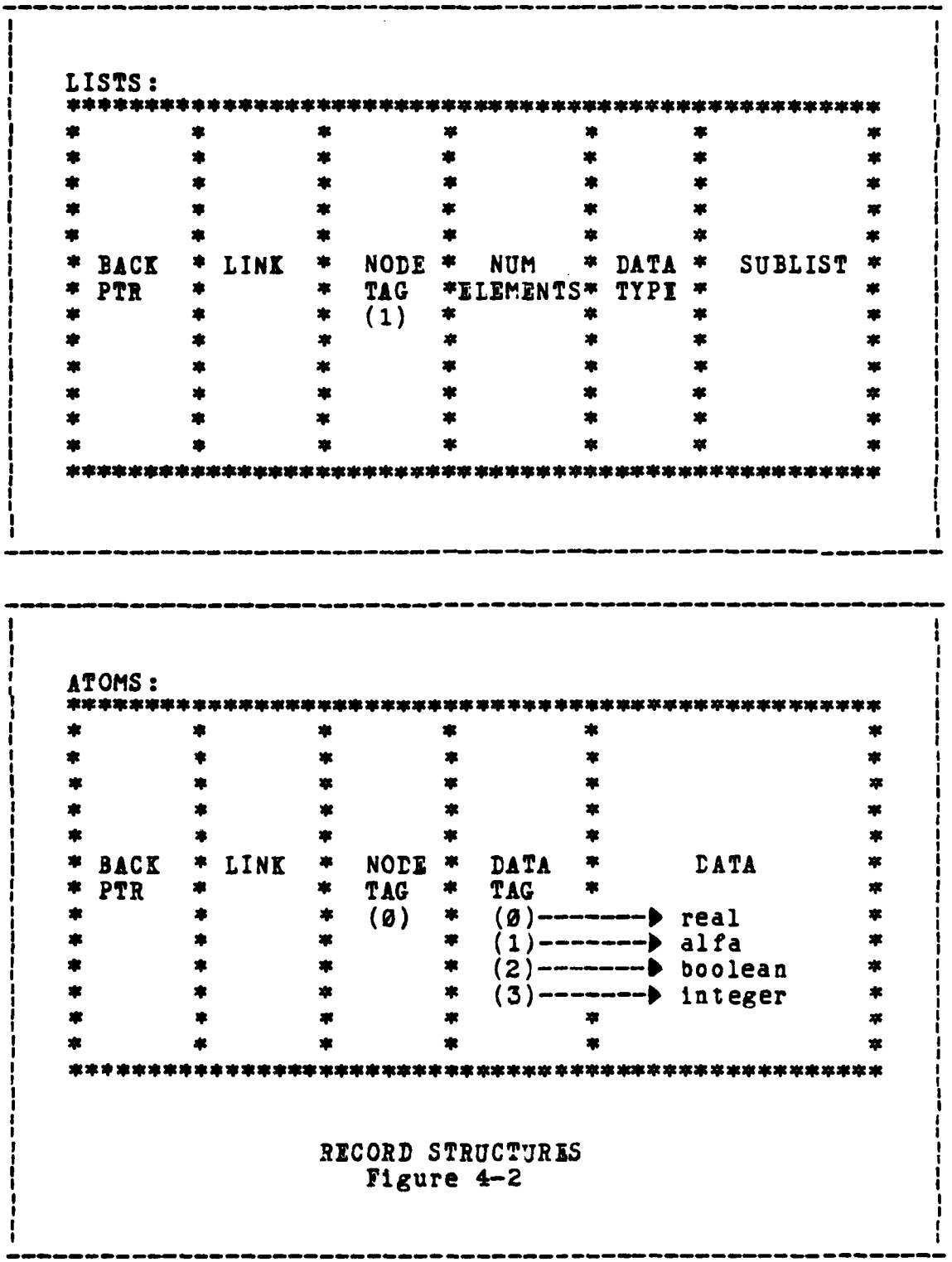
A NODETAG field of 1 indicates that the current element is a list or sequence and that the record is the head node of the list. The remaining parts of the variant record for the "tag 1" case consist of three pointer fields, a data type field and a counter field. The first pointer field, LINK, provides the location of the next non-sequence element (if any) in the current sequence. The second pointer field, BACKPTR, provides the link to the previous record, thus building the doubly linked list structure. The third pointer field, SUBLIST, is the complement of the LINK pointer and provides the location of the first element (if any) in the subsequence. The data type field, DATATYPE, indicates whether the elements in a sequence are alphanumeric or numeric and the counter field, NUMELEMENTS, records the number of elements in a sequence at each level.

A NODETAG field of Ø, indicates that the current element is data. The remaining parts of the variant record for the "tag Ø" case, consist of two pointer fields, a data tag field and a data field. The first pointer field, LINK, provides the location of the next data item in the lowest level sublist. The second pointer field, BACKPTR, provides

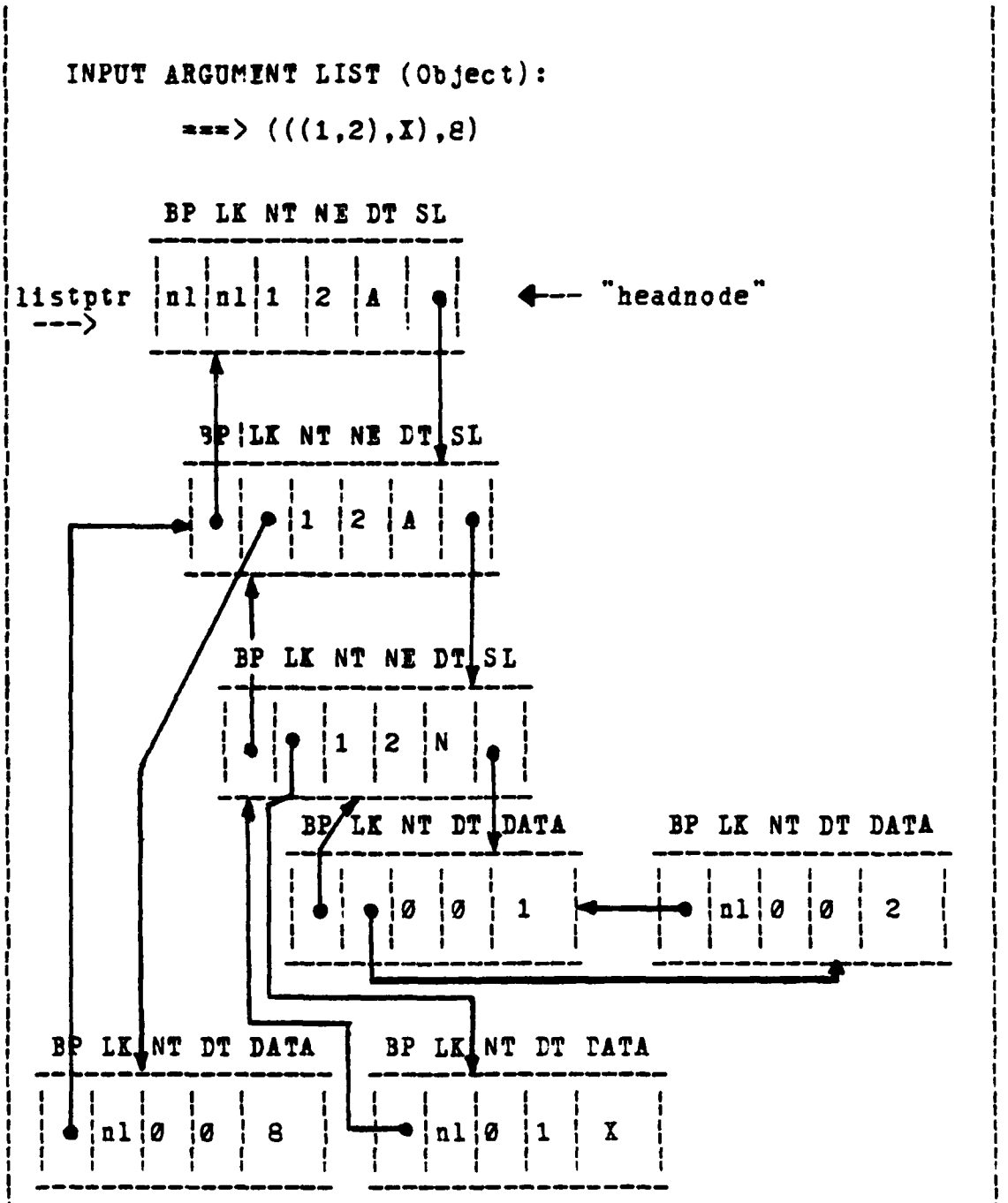
the link to the previous node in the sequence. The data tag field, DATATAG, specifies whether the data item is of type real or integer for numeric operations, type alfa for single and multiple character representations, or type boolean for boolean operations. The data field contains the atoms in each sequence. Figure 4-2 illustrates the two types of record variants that can be generated in the Functional PASCAL System. After the entire data structure is created, a pointer variable, LISTPTR, is returned by the function in order to provide future access by the system and user-defined functions. The very first node created is termed the "head" node and contains critical summary-type administrative information regarding the argument list as a whole. An example of an input argument list and the subsequent data structure created by the input function is provided in Figure 4-3. The source code for the procedure GENLIST is listed in Appendix A.

3. Function Application

Sections 1 and 2 above refer to the "object" portion of a functional programming operation. As illustrated in Figure 4-1, there is also a "function" portion which consists of the system and user-defined functions. The principle of function application mandates that these functions be used to manipulate the input data to achieve the desired output. A complete set of algorithms for the system-defined functions is provided in the latter half of this chapter. The function algorithms are divided into two



RECORD STRUCTURES
Figure 4-2



DATA REPRESENTATION OF AN ARGUMENT LIST
Figure 4-3

catagories; (1) primitives and (2) functional forms. The documentation for each function also includes the input and output parameters, the constraints associated with each function application and numerous practical examples.

4. Output Function

The Functional PASCAL System provides the user with the capability of obtaining an output of any functional programming operation. A functional operation can result in one of three possible outputs; (1) an atom, (2) a sequence or (3) an undefined entity (error condition). When the output is in the form of an atom, there are four possible responses that can be generated; (1) a real number, (2) an integer, (3) a boolean expression or (4) a character. The principle of "information hiding" provides the programmer with the knowledge that irrespective of the function operation performed, the entity returned is a pointer variable to a list structure containing the appropriate response.

In the event that the output is an atom, the returned value is a pointer to a single node containing the respective output data. The type of data in the data field is specified by a data tag of 0, 1, 2 or 3 to indicate if the output is a real number, integer, boolean expression or character, whichever is applicable. If the output is a sequence, the returned value is a pointer to a complete list structure representing the result of the functional operation.

The procedure, WRITELIST, is a generalized system routine that writes a user's output either interactively or in hard copy form. It uses the pointer variable returned from the function operation to accomplish this task. In the case of a sequence, a subordinate procedure, GENWRITE, recursively reassembles the resultant list structure inserting commas and parentheses as required.

In order to provide the critical interface from the Functional to the Conventional Mode, a series of eight additional functions are introduced. To accomplish the task of data extraction from a list structure, the Functional PASCAL System provides four system-defined functions; (1) GETREAL, (2) GETINT, (3) GETBOOL and (4) GETALFA. These functions take a pointer to the list structure containing the output value and return this value to a variable of the same type in the conventional portion of the program. This allows the user to take the results of the function operations in the Functional Mode and transfer them back to the Conventional Mode within the Functional PASCAL program. The process can be reversed in a similar manner. To return a particular data item to the appropriate list structure, the system provides four additional functions; (1) PUTREAL, (2) PUTINT, (3) PUTBOOL and (4) PUTALFA. A pointer variable is, again returned, recreating the identical structure as it existed prior to the call of one of the GET functions. The source code for all of the system output functions is listed in Appendix A.

C. CONVENTIONAL MODE DESIGN

In order to effectively interface the Functional Mode with the Conventional Mode, it is necessary to use the "include" function provided by the UC Berkeley PASCAL compiler. Separate files are maintained for the variables, types and functions related to the Functional Mode of the proposed system in order to avoid conflict with the particular implementation of the UC Berkeley compiler. These files comprise the "system library" and the programmer must incorporate them into the main program by using the "include" function. A detailed description and step-by-step procedure for using the function is provided in Appendix B (User's Manual).

D. SYSTEM-DEFINED FUNCTION ALGORITHMS

Catagory I: Primitives

NAME: Selector

PURPOSE: To select a pre-determined element from a list.

INPUT PARAMETERS: x: A pointer variable to the first node of the list structure

OUTPUT PARAMETERS: x: A pointer to the newly-created list structure containing the element selected

CONSTRAINTS: (1) Input parameter, x, must be a pointer to a two-element list.

(2) The first element of the list must be a positive integer value, s, which represents the element to be selected.

(3) The second element of the list must be a sequence and the number of elements, n, in the sequence must be greater than or equal to s.

FUNCTION CALL: SELECT (x:ptr) : ptr;

EXAMPLE: SELECT (x) where x = (3,((1,2,3),(4.6),c,(3,4)))
===> c

SELECT (x) where x = (4,a,b,c,(d,e))
===> (d,e)

ALGORITHM: SELECT (x) ===>

IF x = (s,(x[1],...,x[n])) AND n >= s THEN

x[s]

ELSE

undefined

ENDIF

SYSTEM-DEFINED FUNCTION ALGORITHMS

Catagory I: Primitives

NAME: Tail

PURPOSE: To disgard the first element of a list and return a list containing the rest of the elements of the list.

INPUT PARAMETERS: x: A pointer variable to the first node of the list structure

OUTPUT PARAMETERS: x: A pointer to the newly-created list structure containing the remaining elements in the list

CONSTRAINTS: (1) Input parameter, x, must be a pointer to a list of n elements where n is greater than or equal to 1.

FUNCTION CALL: TAIL (x:ptr) : ptr;

EXAMPLE: TAIL (x) where x = (1,2,3) ===> (2,3)

```
ALGORITHM:  TAIL (x) ===>
            IF x = (x[1]) THEN
                 empty set
            ELSE IF x = (x[1],...x[n]) AND n >= 2 THEN
                 (x[2],...x[n])
            ELSE
                 undefined
            ENDIF
```

SYSTEM-DEFINED FUNCTION ALGORITHMS

Catagory I: Primitives

NAME: Atom

PURPOSE: To identify whether an object, x, is an atom or a list.

INPUT PARAMETERS: x: A pointer variable to the first node of a list structure

OUTPUT PARAMETERS: x: A pointer variable to a list structure containing the boolean results of the test

CONSTRAINTS: (1) Input parameter, x, must be a pointer to a list of n elements where n is greater than or equal to Ø.

FUNCTION CALL: ATOM (x:ptr) : ptr;

EXAMPLE: ATOM (x) where x = (1,2,3,4) ===> false

ATOM (x) where x = 2 ===> true

```
ALGORITHM:  ATOM (x) ===>
            IF (x = atom) THEN
                 true
            ELSE IF (x <> undefined) THEN
                 false
            ELSE
                 undefined
            ENDIF
```

SYSTEM-DEFINED FUNCTION ALGORITHMS

Catagory I: Primitives

NAME: Identity

PURPOSE: To return the identity of an object, which is the object itself.

INPUT PARAMETERS: x: A pointer variable to the first node in the list structure

OUTPUT PARAMETERS: x: A pointer to the newly-created list structure containing the original object

CONSTRAINTS: (1) Input parameter, x, must be a pointer to a list of n elements where n is greater than or equal to 0.

FUNCTION CALL: ID (x:ptr) : ptr;

EXAMPLE: ID (x) where x = (6,7,(2,3)) ===> (6,7,(2,3))

ID (x) where x = A33 ===> A33

```
ALGORITEM:  ID (x) ===>
            IF (x = atom) THEN
                 atom
            ELSE IF x = (x[1],...,x[n]) THEN
                 (x[1],...,x[n])
            ELSE IF x = ( ) THEN
                 empty set
            ELSE
                 undefined
            ENDIF
```

SYSTEM-DEFINED FUNCTION ALGORITHMS

Catagory I: Primitives

NAME: Equals

PURPOSE: To determine whether the elements of a two-element list are equal.

INPUT PARAMETERS: x: A pointer variable to the first node in the list structure

OUTPUT PARAMETERS: x: A pointer variable to a list structure containing the boolean results of the test

CONSTRAINTS: (1) Input parameter, x, must be a pointer to a two-element list.

FUNCTION CALL: EQUALS (x:ptr) : ptr;

EXAMPLE: EQUALS (x) where x = ((1,2,3),(1,2,3)) ===> true

EQUALS (x) where x = ((1,2,3),(1,2,4)) ===> false

EQUALS (x) where x = (A8,AB) ===> true

```
ALGORITHM:  EQUALS (x) ===>
            IF (x is a two element list) THEN
                 IF (both elements are equal) THEN
                      true
                 ELSE
                      false
                 ENDIF
            ELSE
                 undefined
            ENDIF
```

SYSTEM-DEFINED FUNCTION ALGORITHMS

Catagory I: Primitives

NAME: Null

PURPOSE: To determine if a list contains any elements.

INPUT PARAMETERS: x: A pointer variable to the first node of the list structure

OUTPUT PARAMETERS: x: A pointer variable to a list structure containing the boolean results of the test

CONSTRAINTS: (1) Input parameter, x, must be a pointer to a list of n elements where n is greater than or equal to 0.

FUNCTION CALL: NULL (x:ptr) : ptr;

EXAMPLE: NULL (x) where x = ( ) ===> true

NULL (x) where x = (AB) ===> false

```
ALGORITHM:  NULL (x) ===>
              IF (x = empty set) THEN
                   true
              ELSE IF (x <> undefined) THEN
                   false
              ELSE
                   undefined
              ENDIF
```

SYSTEM-DEFINED FUNCTION ALGORITHMS

Catagory I: Primitives

NAME: Reverse

PURPOSE: To reverse the elements of a list.

INPUT PARAMETERS: x: A pointer variable to the first node of the list structure

OUTPUT PARAMETERS: x: A pointer to the newly-created list structure containing the elements of the original list in reverse order

CONSTRAINTS: (1) Input parameter, x, must be a pointer to a list of n elements where n is greater than or equal to 0.

FUNCTION CALL: REV (x:ptr) : ptr;

EXAMPLE: REV (x) where x = (1,2,3,4) ===> (4,3,2,1)

REV (x) where x = ((1,2),B,3) ===> (3,B,(1,2))

```
ALGORITHM:  REV (x) ===>
            IF x = (x[1],...,x[n]) THEN
                 (x[n],...,x[1])
            ELSE IF x = ( ) THEN
                 empty set
            ELSE
                 undefined
            ENDIF
```

SYSTEM-DEFINED FUNCTION ALGORITHMS

Catagory I: Primitives

NAME: Length

PURPOSE: To determine the number of elements in a list.

INPUT PARAMETERS: x: A pointer variable to the first node of the list structure

OUTPUT PARAMETERS: x: A pointer variable to a list structure containing the number of elements in the list

CONSTRAINTS: (1) Input parameter, x, must be a pointer to a list of n elements where n is greater than or equal to 0.

FUNCTION CALL: LENGTH (x:ptr) : ptr;

EXAMPLE: LENGTH (x) where x = (1,2,3) ===> 3

LENGTH (x) where x = ( ) ===> 0

LENGTH (x) where x = (A,(1,2),(3,4),B) ===> 4

```
ALGORITHM:  LENGTH (x) ===>
              IF x = (x[1],...,x[n]) THEN
                   n
              ELSE IF x = ( ) THEN
                   0
              ELSE
                   undefined
              ENDIF
```

SYSTEM-DEFINED FUNCTION ALGORITHMS

Catagory I: Primitives

NAME: Add

PURPOSE: To provide the arithmetic operation of addition on a two-element list.

INPUT PARAMETERS: x: A pointer variable to the first node of the list structure

OUTPUT PARAMETERS: x: A pointer variable to a list structure containing the sum of the two numbers in the input list

CONSTRAINTS: (1) Input parameter, x, must be a pointer to a two-element list.

(2) Both elements in the two-list must be numbers.

FUNCTION CALL: ADD (x:ptr) : ptr;

EXAMPLE: ADD (x) where x = (1,2) ===> 3

ADD (x) where x = (8,4) ===> 12

```
ALGORITHM:  ADD (x) ===>
            IF ( x = (a,b) ) AND ( a,b are numbers ) THEN
                 ( a + b )
            ELSE
                 undefined
            ENDIF
```

SYSTEM-DEFINED FUNCTION ALGORITHMS

Catagory I: Primitives

NAME: Subtract

PURPOSE: To provide the arithmetic operation of subtraction on a two-element list.

INPUT PARAMETERS: x: A pointer variable to the first node of the list structure

OUTPUT PARAMETERS: x: A pointer variable to a list structure containing the difference between the two numbers in the input list

CONSTRAINTS: (1) Input parameter, x, must be a pointer to a two-element list.

(2) Both elements in the two-list must be numbers.

FUNCTION CALL: SUB (x:ptr) : ptr;

EXAMPLE: SUB (x) where x = (15,5) ===> 10

SUB (x) where x = (3.5,6.5) ===> -3

```
ALGORITHM: SUB (x) ===>
           IF (x is a two-element list) AND
                        (both elements are numbers) THEN
                (difference between two numbers)
           ELSE
                undefined
           ENDIF
```

SYSTEM-DEFINED FUNCTION ALGORITHMS

Catagory I: Primitives

NAME: Multiply

PURPOSE: To provide the arithetic operation of multiplication on a two-element list.

INPUT PARAMETERS: x: A pointer variable to the first node of the list structure

OUTPUT PARAMETERS: x: A pointer variable to the list structure containing the product of the two numbers in the input list

CONSTRAINTS: (1) Input parameter, x, must be a pointer to a two-element list.

(2) Both elements in the two-list must be numbers.

FUNCTION CALL: MUL (x:ptr) : real;

EXAMPLE: MUL (x) where x = (5,9) ===> 45

MUL (x) where x = (-6.2,4) ===> -24.8

```
ALGORITHM:  MUL (x) ===>
            IF (x is a two-element list) AND
                                (both elements are numbers) THEN
                 (product of two numbers)
            ELSE
                 undefined
            ENDIF
```

SYSTEM-DEFINED FUNCTION ALGORITHMS

Catagory I: Primitives

NAME: Divide

PURPOSE: To provide the arithmetic operation of division on a two-element list.

INPUT PARAMETERS: x: A pointer variable to the first node of the list structure

OUTPUT PARAMETERS: x: A pointer variable to the list structure containing the quotient of the two numbers in the input list

CONSTRAINTS: (1) Input parameter, x, must be a pointer to a two-element list.

(2) Both elements in the two-list must be numbers.

(3) The second number in the list cannot be Ø.

FUNCTION CALL: DIV (x:ptr) : ptr;

EXAMPLE: DIV (x) where x = (16,2) ===> 8

DIV (x) where x = (11,2) ===> 5.5

```
ALGORITHM:  DIV (x) ===>
            IF (x is a two-element list) AND
                        (both elements are numbers) AND
                        (the second number <> Ø) THEN
                 (quotient of two numbers)
            ELSE
                 undefined
            ENDIF
```

SYSTEM-DEFINED FUNCTION ALGORITHMS

Catagory I: Primitives

NAME: Transpose

PURPOSE: To transpose a list of n elements containing m elements into a list of m elements containing n elements.

INPUT PARAMETERS: x: A pointer variable to the first node of the list structure

OUTPUT PARAMETERS: x: A pointer to the newly-created list structure representing the transposed elements of the original list

CONSTRAINTS: (1) Input parameter, x, must be a pointer to a list of n elements where n is greater than or equal to 2.

(2) Each element in the list of n elements must contain exactly m elements where m is greater than or equal to 0.

(3) x[i] = (x[i1],...,x[im]) (Input Element)
y[j] = (x[1j],...,x[nj]) (Output Element)

(4) 1 =< i =< n and 1 =< j =< n

FUNCTION CALL: TRANS (x:ptr) : ptr;

EXAMPLE: TRANS (x) where x = ((1,2),(3,4),(5,6))
===> ((1,3,5),(2,4,6))

ALGORITHM: TRANS (x) ===>

```
IF x = (( ),...,( )) THEN
     empty set
ELSE IF x = (x[1],...,x[n]) THEN
     (y[1],...,y[m])
ELSE
     undefined
ENDIF
```

SYSTEM-DEFINED FUNCTION ALGORITHMS

Catagory II: Functionals

NAME: Insert

PURPOSE: To perform selected binary function operations on an entire list structure beginning with the innermost level (right to left) and using the result as input to the next higher level until the outermost level is reached and the final result obtained.

INPUT PARAMETERS: x: A pointer variable to the first node of the list structure

f: The selected binary function to be inserted into the list structure

OUTPUT PARAMETERS: x: A real number value representing the final result of the insert operation for the selected function

CONSTRAINTS: (1) Input parameter, x, must be a pointer to a list of n elements where n is greater than or equal to 0.

(2) The selected primitive function operation must be compatible with the elements of the list.

FUNCTION CALL: INSERT(function f(x:ptr) : ptr; x:ptr) : ptr;

EXAMPLE: INSERT (f,x) where f = ADD and x = (8,10,12,14)
===> ADD (8,ADD (10,ADD (12,ADD (14))))
===> ADD (8,ADD (10,ADD (12,14)))
===> ADD (8,ADD (10,26))
===> ADD (8,36)
===> 44

INSERT (f,x) where f = MULTIPLY and x = (2,6,8)
===> MUL (2,MUL (6,MUL (8)))
===> MUL (2,MUL (6,8))
===> MUL (2,48)
===> 96

```
INSERT (f,x) where f = SUBTRACT and x = ( )
===>  SUB ( )
===>  Ø

INSERT (f,x) where f = SUBTRACT and x = (5)
===>  SUB (5)
===>  5
```

```
ALGORITHM:  INSERT (f,x) ===>

            IF x = (x[1]) THEN

                 x[1]

            ELSE IF x = (x[1],...,x[n]) and n >= 2 THEN

                 f (x[1], INSERT (f, (x[2],...,x[n])))

            ELSE IF x = ( ) THEN

                 Ø

            ELSE

                 undefined

            ENDIF
```

SYSTEM-DEFINED FUNCTION ALGORITHMS

Catagory II: Functionals

NAME: Apply to All

PURPOSE: To perform a selected function operation on each element in a list structure.

INPUT PARAMETERS: x: A pointer variable to the first node of the list structure

f: The selected function to be applied to each element in the list structure

OUTPUT PARAMETERS: x: A pointer to the newly-created list structure representing the result of the application of a function to each element of the list structure

CONSTRAINTS: (1) Input parameter, x, must be a pointer to a list of n elements where n is greater than or equal to Ø.

FUNCTION CALL: APTALL (*function f*(x:ptr) : ptr; x:ptr) : ptr;

EXAMPLE: APTALL (f,x) where x = ((1,2),(3,4),(5,6)) and f = ADD
===> (ADD (1,2), ADD (3,4), ADD (5,6))
===> (3,7,11)

APTALL (f,x) where x = ((A,B,C),(1,2,3,4)) and f = TAIL
===> (TAIL (A,B,C), TAIL (1,2,3,4))
===> ((B,C),(2,3,4))

APTALL (f,x) where x = ((2,4,6),(1,2,3,4),(A,B)) and f = REVERSE
===> (REV (2,4,6), REV (1,2,3,4), REV (A,B))
===> ((6,4,2),(4,3,2,1),(B,A))

APTALL (f,x) where x = ((A,B,(1,2),(10,12),( )) and f = LENGTH
===> (LENGTH (A,B,(1,2),LENGTH (10,12),LENGTH ( )
===> (3,2,Ø)

```
ALGORITHM:  APTALL (f,x) ===>
            IF x = ( ) THEN
                 empty set
            ELSE IF x = (x[1],...,x[n]) THEN
                 (f (x[1]),...,f (x[n]))
            ELSE
                 undefined
            ENDIF
```

V. SYSTEM IMPLEMENTATION

A. IMPLEMENTATION METHODOLOGY

The implementation of the Functional PASCAL System (Phase I) includes the coding, testing and debugging of all component parts of the system and the integration of these components into a conventional programming environment. Due to the strict use of the system development principles of "top-down design" and "information hiding", individual modules can be constructed independently. The detailed input and output parameters specified for the library subroutines, as well as the constraints placed upon each of the modules, enable the developers to make realistic and valid assumptions about the independent segments of the program. The independent components, being essentially "black boxes", can be fully coded, tested and debugged separately before interfacing with the entire system. The system implementation plan for the Functional PASCAL System is divided into three distinct phases:

I -- Coding, testing and debugging of independent system-defined modules (System Sub-Mode)

II -- Testing of prototype user-defined functions (User Sub-Mode)

III -- Full system integration of Functional and Conventional Modes of operation

B. CODING AND CHECK-OUT

The interactive input function, READLIST, is the first module to become fully operational within the Functional Mode of the system. It is the most critical portion of the system, as it takes raw data from the user in the form of an "object", makes format correctness decisions and either creates an appropriate data representation or issues pertinent error messages. The importance of the input function cannot be overemphasized, as it not only formats the data for the user, but also calculates and records vital administrative information required for future function operations in the program.

One of the key modules used by the input function is the data structure creation procedure, GENLIST. Operating on the input string, this procedure removes all of the "syntactic sugar", such as parentheses and commas, before assembling the final representation. The successful completion of an interactive read operation results in the function returning to the main program, a pointer variable in order to provide future access to the user's formatted object.

The output functions are the next logical components to be brought on-line. These functions return to the user the desired response in the form of a real number, integer, character, boolean expression or sequence, depending on the primitive or functional invoked on the "object". The output procedure, WRITELIST, in the case of a sequence, reinserts any parentheses or commas to provide the user with a

response in the same format as outlined in the User's Manual (Appendix B).

With the completion of the input and output functions, the focus of the development is shifted from the "objects" to the "function modules" which operate on them. The primitive functions are separated into two developmental groups: those that produce atoms as output and those that produce sequences as output. Each primitive is coded, tested and debugged using the interactive input function as the source of its input and the output function for the output from list construction operations. After the successful testing of the primitive functions, the more complex functional forms are incorporated into the system. A complete source code listing for all system-defined functions is provided in Appendix A. A summary of the test procedures and results is included in Appendix C.

The development and testing of the individual system-defined functions, above, is only the first step in the implementation plan for the total Functional PASCAL System. The ability of users to define their own functions is paramount to the overall success of the system, providing both flexibility and power. The user-defined functions can be implemented by defining new functions within the main conventional program, following the syntactic rules required by UC Berkeley PASCAL. The functions can be created using any of the system-defined functions resident in the library. It is the responsibility of the programmer to carefully plan

and monitor all appropriate inputs and outputs of the individual components when creating new functions. Sufficient error checking is included within the system to protect the user from invalid operations, i.e. operations which violate the expected input and output parmeters or constraints on the modules. However, the overall result of such errors is the same as would be expected from the UC Berkeley compiler--termination of execution and the display of an error message. Some examples of simple user-defined functions are listed in Appendix C.

The third and final phase of the implementation plan is "system integration". Full "system integration" is accomplished by constructing a simple PASCAL program (Conventional Mode) which utilizes the newly-created Functional Mode at various points in the application. The use of the functional components at designated locations in the conventional program illustrates, in a real-world environment, the movement along the "programming language continuum" discussed in Chapter III. An example of a Functional PASCAL program is provided in Appendix C.

C. TESTING

1. Testing Strategy

The software testing strategy developed for the Functional PASCAL System involves both the "unit testing" and "integration testing" philosophies [8]. The principle of "unit testing" is used to comprehensively test each of the system-defined functions that are to reside in the system

library. "Unit testing" focuses on the actual intricacies of the code within each module. It is at this point that software validation occurs to insure that each function operates correctly and contains the features prescribed by its requirements and specifications. The testing at the "unit" level emphasizes verification of logic, computations and data handling [8].

After the successful testing of each of the system-defined functions, the principle of "integration testing" is invoked. The "integration testing" views the system at the component or module level rather than at the detailed level of code as in "unit testing". This level of testing considers the interaction between software modules and the overall operation of the Functional PASCAL System. It is at this point in the testing process that subtle errors emerge due to the complex interactions between components.

2. Error Handling Procedures

The Functional PASCAL System provides two basic levels of error checking. The first level occurs upon entry of an "object" into the system via READLIST. Improperly structured input strings, i.e. missing or misplaced parentheses or commas, result in an error message indicating a "malformed object". The program is terminated immediately by the use of a PASCAL GOTO statement which transfers control back to the main program from the location at which the error is detected. The second level of error checking

occurs upon the application of a system or user-defined function to an "object". Inappropriate or incompatible function operations attempted on an input string result in an error message indicating the nature of the problem and, as in level one errors, termination of program execution. Both level one and level two errors are fatal and constitute a non-recoverable operation. A complete listing of possible error conditions and corrective procedures is provided in Appendix B (User's Manual).

VI. CONCLUSIONS

The success of any endeavor can be measured by the extent to which the initial goals and objectives are obtained. In this thesis, the hypothesis that the theory of pure functional programming can be applied to a standard conventional programming language is tested. The absence of a "working" functional programming language that the average conventional language programmer can use and understand prompted the initial investigation of this idea. The findings of the researchers are conclusive; the resultant design, development and implementation of a simple, yet powerful functional programming system operating within a conventional environment positively reinforce the original hypothesis.

The primary consideration in the design of the Functional PASCAL System is to provide the user with a unique opportunity to experiment with the concepts of functional programming in the more traditional, "imperative" language environment. A secondary consideration is to build a simple, "user friendly" system that is not only easy to understand but easy to operate. The user is given a clear-cut choice as to which programming methodology best suits the particular application and can react accordingly. It is the integration of these functional programming concepts into a conventional, high-level program that allows

the user to gain an added dimension of flexibility in complex problem-solving situations. In addition, the implementation of the Functional PASCAL System bridges the "applicative language gap" which now exists in the predominently conventional language computer industry by giving the user the ability to gradually become accustomed to a new form of programming.

The efficiency of implementation for the Functional PASCAL System is not specifically addressed within the scope of the study. The liberal use of memory within the system is justified by its rapidly decreasing cost and increased benefits in developing a "user friendly" system. The greatly escalating costs of programmer time in the areas of development and maintenance give license to be less concerned than in the past with efficiency considerations. Programmer productivity and the optimization of human resources is not only the dominant factor in business and industry today, but also the motivation behind this thesis.

All in all, the Functional PASCAL System enables the user to move freely between two distinct programming methodologies. The total transparency to the user of the many intracacies of functional programming operations results in a simple, yet powerfully flexible system complete with error checking and diagnostics.

The scope of the study (Phase I) is limited to the development of only about one-half of the primitive functions and functional forms within the Backus system. The

Functional PASCAL System, therefore, is flexible enough to accept additional system-defined modules at any time. It is envisioned that Phase II of the process would involve the design and development of the remaining primitives and functionals to further enhance the overall system.

Additional research is needed in the area of operational efficiency in order to evaluate the expense of implementing a functional system. Specific areas of interest are response time, execution time and total memory requirements. Exploration into the possibility of integrating the Backus functional programming system into other high-level languages may also prove valuable. Finally, the extension of the Functional PASCAL System into a concurrent programming environment would be an excellent topic for follow-on research and present the language designers with even greater challenges in the future.

```
{**C**  THIS INCLUDE FILE ("label.i") ASSIGNS A LABEL (99) AT THE
        END OF THE USER'S PROGRAM, TO WHICH ANY FUNCTION OR PRO-
        CEDURE IN THE FUNCTIONAL PASCAL PROGRAM DETECTING AN ERROR
        CAN JUMP.  THIS FILE IS INCLUDED IN THE USER'S PROGRAM BY
        REQUIRING HIM TO USE THE FOLLOWING STATEMENT BEFORE THE
        "TYPE" DECLARATIONS:

              program compute(input,output,outfile);

              # include "label.i" (NOTE:  # MUST BE LEFT JUSTIFIED)

                type
                  .
                  .
                  .                                                  }


label 99;
```

```
{**C**  THIS INCLUDE FILE ("type.i) CONTAINS ALL THE TYPE DECLARATIONS
        NEEDED IN THE USER'S PROGRAM TO ALLOW THE FUNCTIONAL PASCAL
        PORTION TO WORK.  THIS FILE IS INCLUDED IN THE USER'S PROGRAM
        BY REQUIRING HIM TO USE THE FOLLOWING STATEMENTS AFTER HIS
        TYPE DECLARATIONS:

              type
        #       include "type.i" (NOTE: THE # MUST BE LEFT JUSTIFIED)}



    tagtype             = 0..3;
    ptr                 = ^node;                    {POINTER TO A RECORD}
    node                = record                    {RECORD DEFINITION*}
                            backptr,link:ptr;
                            case nodetag:tagtype of
                              0:  (case datatag:tagtype of
                                     0:  (num:real);
                                     1:  (ident:alfa);     {SEE COMMENT}
                                     2:  (bool:boolean);
                                     3:  (int:integer));
                              1:  (numelements:integer;       {BELOW}
                                   datatype:char;
                                   sublist:ptr)
                          end;
```

```
{**C**  THIS INCLUDE FILE ("var.i") CONTAINS ALL THE VARIABLE DEC-
        LARATIONS NEEDED IN THE USER'S PROGRAM TO ALLOW THE FUNC-
        TIONAL PASCAL PORTION TO WORK.  THIS FILE IS INCLUDED IN
        THE USER'S PROGRAM BY REQUIRING HIM TO USE THE FOLLOWING
        STATEMENTS AFTER HIS VARIABLE DECLARATIONS:

              var
            #   include "var.i" (NOTE: THE # MUST BE LEFT JUSTIFIED)}



    outfile                 :  text;        {OUTPUT BUFFER}
    error                   :  boolean;     {DETECTS ERROR CONDITION}
```

```
{**C**  THIS INCLUDE FILE ("init.i") CONTAINS ALL THE INITIALIZATION
        STATEMENTS NEEDED IN THE USER'S PROGRAM TO ALLOW THE FUNCTION-
        AL PASCAL PORTION TO WORK.  THIS FILE IS INCLUDED IN THE
        USER'S PROGRAM BY REQUIRING HIM TO USE THE FOLLOWING STATE-
        MENTS AFTER THE "BEGIN" IN HIS MAIN PROGRAM:

              begin
        #       include "init.i" (NOTE: THE # MUST BE LEFT JUSTIFIED)}



    rewrite(outfile);                   {ZERO OUTFILE}
    error := false;                     {ERROR FLAG OFF}
```

```
{**C**  THIS INCLUDE FILE ("errorlabel.i") CONTAINS THE LABEL "99"
        TO WHICH ANY FUNCTIONAL PASCAL FUNCTION OR PROCEDURE DETECT-
        ING AN ERROR CAN JUMP TO EXIT THE PROGRAM AND GIVE AN ERROR
        MESSAGE.  THIS FILE IS INCLUDED AS THE LAST STATEMENT IN THE
        USER'S PROGRAM BEFORE THE "END":

             begin
               .
               .
               .
           #    include "errorlabel.i" (NOTE: # MUST BE LEFT JUSTIFIED)
             end.  (MAIN PROGRAM)}
```



```
99:  if error then begin
       writeln;writeln;writeln(outfile);writeln(outfile);
       writeln('          execution terminated.');writeln;
       writeln(outfile,'          execution terminated.');
       writeln(outfile);
     end
```

```
{**********************************************************************}
{**********************************************************************}
{**                                                                  **}
{**                                                                  **}
{**           FUNCTIONAL PASCAL FUNCTIONS AND PROCEDURES             **}
{**                                                                  **}
{**                                                                  **}
{**********************************************************************}
{**********************************************************************}


{**C**  THIS INCLUDE FILE ("functions.i") CONTAINS ALL THE FUNCTIONS
        AND PROCEDURES NECESSARY IN THE USER'S PROGRAM TO ALLOW THE
        FUNCTIONAL PASCAL PORTION TO WORK.  THIS FILE IS INCLUDED IN
        THE USER'S PROGRAM BY REQUIRING HIM TO USE THE FOLLOWING
        STATEMENT AFTER HIS "DECLARATION SECTION" AND BEFORE HIS
        "MAIN PROGRAM":

              var
                .
                .
                x : real;

            # include "functions.i" (NOTE: # MUST BE LEFT JUSTIFIED)

              begin  (Main Program)}
```

```
{*************************************************************************}
{*************************************************************************}

function READLIST  :  ptr;


{**C**  THIS FUNCTION INTERACTIVELY READS IN THE ARGUMENT LIST
        (OBJECT) OF A FUNCTIONAL PROGRAMMING SYSTEM THE USER HAS
        DEFINED, CREATES A LINKED LIST DATA STRUCTURE TO REPRESENT
        THIS LIST, AND RETURNS A POINTER TO THIS DATA STRUCTURE.}

{*************************************************************************}
{*************************************************************************}

  type
    buffer              =  array [1..80] of char;    {OBJECT BUFFER}
  var
    listbuffer          :  buffer;
    listptr             :  ptr;
    packit              :  alfa;
    alphaflag           :  boolean;
    length,i            :  integer;
    character           :  char;
```

```
{*************************************************************************}

function FINDNUM  :  real;

{**C**  THIS FUNCTION TAKES THE NEXT SINGLE OR MULTIPLE DIGIT NUMBER
        STORED IN THE ARRAY "LISTBUFFER" AND PACKS THEM INTO A VAR-
        IABLE OF TYPE "REAL" SO THAT THEY CAN BE ENTERED INTO THE
        DATA FIELD OF A RECORD.}

{*************************************************************************}

  var integerpart       :  integer;
      r,realpart        :  real;
      nflag             :  boolean;
  begin
    integerpart := 0;
    realpart := 0.0;
    nflag := false;
    if listbuffer[i] = '-' then begin      {IF NEGATIVE SET FLAG}
      i := i + 1;
      nflag := true
    end;  {if}
    while listbuffer[i] in ['0'..'9'] do begin  {COMPUTE INTEGER PART}
      integerpart := integerpart*10+(ord(listbuffer[i])-ord('0'));
      i := i + 1
    end;  {while}
    if listbuffer[i] = '.' then begin   {COMPUTE REAL PART IF NECESSARY}
      i := i + 1;
      r := 1.0;
      while listbuffer[i] in ['0'..'9'] do begin
        r := r*0.1;
        realpart := realpart+r*(ord(listbuffer[i])-ord('0'));
        i := i + 1
      end;  {while}
      if not(listbuffer[i] in [',',')',';']) then begin   {ERROR}
        writeln;writeln(outfile);
        writeln('Malformed OBJECT:  Excess or missing characters.');
```

```
        write(outfile,'Malformed OBJECT:  Excess or missing ');
        writeln(outfile,'characters.');
        error := true;
        goto 99
      end  {if}
    end  {if}
    else if not(listbuffer[i] in [',',')',';']) then begin  {ERROR}
      writeln;writeln(outfile);
      writeln('Malformed OBJECT:  Excess or missing characters.');
      write(outfile,'Malformed OBJECT:  Excess or missing ');
      writeln(outfile,'characters.');
      error := true;
      goto 99
    end;
    if nflag then                                  {NEGATIVE NUMBER}
      FINDNUM := (-1.0)*(integerpart+realpart)
    else FINDNUM := integerpart+realpart;
  end;  {FINDNUM}
```

```
{*************************************************************************}

procedure FINDIDENT ;

{**C**  THIS PROCEDURE TAKES A SINGLE OR MULTIPLE CHARACTER IDENT-
        IFIER STORED IN THE ARRAY "LISTBUFFER" AND PACKS THEM INTO
        A VARIABLE OF TYPE "ALFA" SO THAT THEY CAN BE ENTERED INTO
        THE DATA FIELD OF A RECORD.}

{*************************************************************************}

  var j                 : integer;
  begin
    for j := 1 to 10 do
      packit[j] := ' ';                          {CLEAR THE PACKED ARRAY}
    j := 1;
    while not(listbuffer[i] in [',',')',';']) do begin {PACK IDENTIFIER}
      packit[j] := listbuffer[i];
      i := i + 1;
      j := j + 1
    end  {while}
  end;  {FINDIDENT}
```

```
{*************************************************************************}

procedure GENLIST (firstnodeptr    :  ptr;
                   var headptr     :  ptr;
                   var listbuffer  :  buffer;
                   var i           :  integer;
                   var count       :  integer;
                   var flag        :  boolean );

{**C**  THIS PROCEDURE RECURSIVELY CREATES A DOUBLY LINKED LIST REP-
        RESENTATION FOR THE USER'S ARGUMENT LIST (OBJECT).}

{*************************************************************************}

  var alphaflag              :  boolean;
      length                 :  integer;
      lastnodeptr,moveptr    :  ptr;
  begin
    i := i + 1;
    length := 0;
    alphaflag := false;
    lastnodeptr := firstnodeptr;

    if listbuffer[i] = ')' then begin                    {EMPTY LIST}
      new(headptr);                                     {CREATE A NODE}
      headptr^.backptr := lastnodeptr;
      headptr^.link := nil;
      headptr^.nodetag := 0;
      headptr^.datatag := 1;
      headptr^.ident := 'none      '
    end
    else if listbuffer[i] in ['(','a'..'z','A'..'Z','0'..'9','-','.']
                                                         {SEQUENCE}
    then begin                                          {CREATE A NODE}
      new(headptr);                                     {ANCHORS HEADPTR}
      moveptr := headptr;
      while not (listbuffer[i] in [')',';']) do begin
        count := count + 1;                        {INCREMENT LIST LENGTH}
```

```
if listbuffer[i] = '(' then begin          {ANOTHER LIST}
  moveptr^.backptr := lastnodeptr;
  moveptr^.nodetag := 1;
  GENLIST(headptr,moveptr^.sublist,listbuffer,i,length,alphaflag);
                                  {RECURSIVELY GENERATE THE LIST}
  moveptr^.numelements := length;
  length := 0;                                  {ZERO LENGTH}
  if alphaflag then moveptr^.datatype := 'A'
  else moveptr^.datatype := 'N';
  i := i + 1
end  {if}
else if listbuffer[i] in ['a'..'z','A'..'Z','0'..'9','-','.']
then begin                                          {ATOM}
  moveptr^.backptr := lastnodeptr;
  moveptr^.nodetag := 0;
  if listbuffer[i] in ['a'..'z','A'..'Z'] then begin {IDENTIFIER}
    flag := true;
    moveptr^.datatag := 1;
    FINDIDENT;
    moveptr^.ident := packit                {FIND IDENTIFIER}
  end  {if}
  else begin                                    {NUMBER}
    moveptr^.datatag := 0;
    moveptr^.num := FINDNUM                   {FIND NUMBER}
  end  {else}
end  {else if}
else begin                                  {ERROR-JUMP TO LABEL}
  writeln;writeln(outfile);
  write('Malformed OBJECT:  ');
  writeln('ATOMS cannot begin with special symbols.');
  write(outfile,'Malformed OBJECT:  ');
  writeln(outfile,'ATOMS cannot begin with special symbols.');
  error := true;
  goto 99
end;  {else}
if listbuffer[i] = ',' then begin             {MORE ELEMENTS}
  i := i + 1;
```

```
        if not(listbuffer[i] in [')',';']) then begin
          new(moveptr^.link);                          {CREATE A NODE}
          lastnodeptr := moveptr;                        {ANCHOR PTR}
          moveptr := moveptr^.link;                  {POINT TO NEXT NODE}
        end  {if}
        else begin                                    {ERROR-JUMP TO LABEL}
          writeln;writeln(outfile);
          writeln('Malformed OBJECT:  Excess Comma.');
          writeln(outfile,'Malformed OBJECT:  Excess Comma.');
          error := true;
          goto 99
        end
      end
      else moveptr^.link := nil;                       {END OF LIST ')'}
    end  {while}
  end  {else if}
  else begin                                          {ERROR-JUMP TO LABEL}
    writeln;writeln(outfile);
    write('Malformed OBJECT:  ');
    writeln('ATOMS cannot begin with special symbols.');
    write(outfile,'Malformed OBJECT:  ');
    writeln(outfile,'ATOMS cannot begin with special symbols.');
    error := true;
    goto 99
  end;  {else}
  if alphaflag or flag then flag := true          {ANY ALPHANUMERICS?}
end;  {GENLIST}
```

```
begin                                                    {**READLIST**}
  for i := 1 to 80 do                                     {INITIALIZE}
    listbuffer[i] := ' ';
  for i := 1 to 10 do
    packit[i] := ' ';
  listptr := nil;
  length := 0;
  alphaflag := false;
  i := 1;

  writeln('Enter OBJECT.');
  write('==>');
  while not eoln do begin
    read(character);                              {READ OBJECT INTO BUFFER}
    if character <> ' ' then begin                   {GET RID OF BLANKS}
      listbuffer[i] := character;
      i := i + 1
    end;  {if}
  end;  {while}
  listbuffer[i] := ';';                {ESTABLISH ';' AS END-OF-BUFFER MARK}
  readln;

  i := 1;
  if listbuffer[i] = '(' then begin {OBJECT IS A SEQUENCE (x1,..,xn)
                                                    OR THE EMPTY SET}

    new(listptr);                                      {CREATE A NODE}
    listptr^.backptr := nil;
    listptr^.nodetag := 1;
    GENLIST(listptr,listptr^.sublist,listbuffer,i,length,alphaflag);
                                          {RECURSIVELY GENERATE THE LIST}
    listptr^.numelements := length;
    if listptr^.numelements = 0 then listptr^.datatype := 'U'
    else if alphaflag then listptr^.datatype := 'A'
    else listptr^.datatype := 'N';
    listptr^.link := nil;
    i := i + 1
```

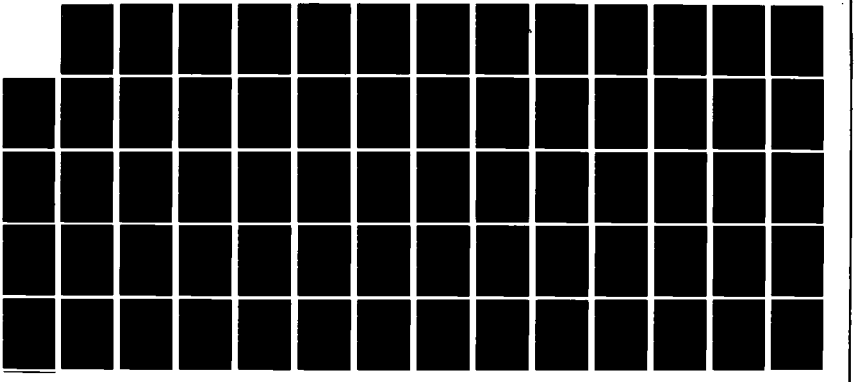
```
    end  {if}
    else if listbuffer[i] in ['a'..'z','A'..'Z','0'..'9','-','.'] then
      begin                                          {OBJECT IS AN ATOM}
        new(listptr);                                  {CREATE A NODE}
        listptr^.backptr := nil;
        listptr^.link := nil;
        listptr^.nodetag := 0;
        if listbuffer[i] in ['a'..'z','A'..'Z'] then begin {IDENTIFIER}
          listptr^.datatag := 1;
          FINDIDENT;
          listptr^.ident := packit                    {FIND IDENTIFIER}
        end  {if}
        else begin                                   {OBJECT IS A NUMBER}
          listptr^.datatag := 0;
          listptr^.num := FINDNUM                       {FIND NUMBER}
        end;  {else}
      end  {else}
    else begin                                       {ERROR-JUMP TO LABEL}
      writeln;writeln(outfile);
      write('Malformed OBJECT:  ');
      writeln('Special symbols not allowed.');
      write(outfile,'Malformed OBJECT:  Special symbols not ');
      writeln(outfile,'allowed.');
      error := true;
      goto 99
    end;  {else}
    if listbuffer[i] <> ';' then begin               {ERROR-JUMP TO LABEL}
      writeln;writeln(outfile);
      writeln('Malformed OBJECT:  Excess or missing characters.');
      writeln(outfile,'Malformed OBJECT:  Excess or missing characters.');
      error := true;
      goto 99
    end;  {if}
    READLIST := listptr;
  end;  {READLIST}
```

```
{*************************************************************************}
{*************************************************************************}

procedure WRITELIST (x : ptr);


{**C**  THIS PROCEDURE RECURSIVELY WRITES OUT TO THE TERMINAL THE
        LIST REPRESENTATION OF THE DATA STRUCTURE CREATED IN ANY
        OF THE PRIMITIVES OR FUNCTIONALS.}

{*************************************************************************}
{*************************************************************************}



{*************************************************************************}

procedure GENWRITE (x : ptr);

{**C**  THIS PROCEDURE RECURSIVELY WRITES A LIST TO THE TERMINAL.}

{*************************************************************************}

  begin
    while x <> nil do begin                  {CONTINUE UNTIL NO ELEMENTS}
      case x^.nodetag of                           {CHECK TAG FIELD}
        0:  case x^.datatag of                  {ATOM-CHECK TYPE FIELD}
              0:  begin                                        {REAL}
                    write(x^.num:0:2);
                    write(outfile,x^.num:0:2)
                  end;
              1:  begin                                        {ALFA}
                    write(x^.ident);
                    write(outfile,x^.ident)
                  end;
              2:  begin                                     {BOOLEAN}
                    write(x^.bool);
```

```
                    write(outfile,x^.bool)
                  end;
              3:  begin                                    {INTEGER}
                    write(x^.int:2);
                    write(outfile,x^.int:2)
                  end
            end;  {case}
        1:  begin                                  {SEQUENCE OR NULL SET}
              if x^.numelements = 0 then begin       {NULL LIST}
                write('()');
                write(outfile,'()')
              end  {if}
              else begin                             {SEQUENCE}
                write('(');
                write(outfile,'(');
                GENWRITE(x^.sublist);        {RECURSIVELY GENERATE LIST}
                write(')');
                write(outfile,')')
              end;  {else}
            end;  {begin}
      end;  {case}
      if x^.link <> nil then begin                {MORE ELEMENTS}
        write(',');
        write(outfile,',')
      end;  {if}
      x := x^.link;                                  {MOVE PTR}
    end  {while}
  end;  {GENWRITE}
```

AD-A120 399 FUNCTIONAL PASCAL: AN INTERIM SOLUTION TO A CHANGING COURSE IN PROGRAMMING LANGUAGE DEVELOPMENT(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA O D BORCHELLER ET AL. 2/2

UNCLASSIFIED JUN 82 F/G 9/2 NL



END

FILMED

DTIC

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
begin                                                    {**WRITELIST**}
  if x = nil then begin                                 {MALFORMED OBJECT}
    writeln;writeln(outfile);
    writeln('Unable to Write List:  Pointer is Nil.');
    writeln(outfile,'Unable to Write List:  Pointer is Nil.');
    error := true;
    goto 99
  end  {if}
  else begin
    case x^.nodetag of                                    {CHECK TAG FIELD}
      0:  case x^.datatag of                            {ATOM-CHECK TYPE FIELD}
            0:  begin                                              {REAL}
                  write(x^.num:0:2);
                  write(outfile,x^.num:0:2)
                end;
            1:  begin                                              {ALFA}
                  write(x^.ident);
                  write(outfile,x^.ident)
                end;
            2:  begin                                             {BOOLEAN}
                  write(x^.bool);
                  write(outfile,x^.bool)
                end;
            3:  begin                                             {INTEGER}
                  write(x^.int:2);
                  write(outfile,x^.int:2)
                end
          end;  {case}
      1:  begin                                        {SEQUENCE OR NULL LIST}
            if x^.numelements = 0 then begin                      {NULL LIST}
              write('()');
              write(outfile,'()')
            end
            else begin                                             {SEQUENCE}
              write('(');
              write(outfile,'(');
              GENWRITE(x^.sublist);      {RECURSIVELY WRITE THE LIST}
```

```
                write(')');
                write(outfile,')')
              end;  {else}
            end  {begin}
        end {case}
      end;  {else}
    end;  {WRITELIST}
```

```
{*************************************************************************}
{*************************************************************************}

function ADD (x:ptr) : ptr;


{**C**  THIS PRIMITIVE FUNCTION TAKES A POINTER TO A LIST WITH TWO
       ELEMENTS, ADDS THE TWO ELEMENTS, CREATES A DATA REPRESEN-
       TATION FOR THE RESULT, AND PASSES A POINTER TO THIS STRUC-
       TURE.}

{*************************************************************************}
{*************************************************************************}

  var realsum  :  real;
      intsum   :  integer;
  begin
    if x^.nodetag = 1 then begin                          {SEQUENCE}
      if (x^.numelements = 2) then begin                {EXACTLY 2 ELEMENTS}
        if (x^.sublist^.nodetag = 0) and
           (x^.sublist^.link^.nodetag = 0) then begin  {BOTH ARE ATOMS}
          if (x^.sublist^.datatag = 0) and
             (x^.sublist^.link^.datatag = 0) then begin {BOTH ARE REALS}
            realsum := x^.sublist^.num + x^.sublist^.link^.num;
            new(x);                                       {CREATE A NODE}
            x^.backptr := nil;
            x^.link := nil;
            x^.nodetag := 0;
            x^.datatag := 0;
            x^.num := realsum;
            ADD := x
          end  {if}
          else if (x^.sublist^.datatag = 3) and
                  (x^.sublist^.link^.datatag = 3) then begin  {INTEGERS}
            intsum := x^.sublist^.int + x^.sublist^.link^.int;
            new(x);                                       {CREATE A NODE}
            x^.backptr := nil;
```

```
            x^.link := nil;
            x^.nodetag := 0;
            x^.datatag := 3;
            x^.int := intsum;
            ADD := x
          end  {if}
          else begin                                    {ERROR-JUMP TO LABEL}
            writeln;writeln(outfile);
            writeln('ADD can only add reals or integers.');
            writeln(outfile,'ADD can only add reals or integers.');
            error := true;
            goto 99
          end;  {else}
        end  {if}
        else begin                                      {ERROR-JUMP TO LABEL}
          writeln;writeln(outfile);
          writeln('ADD must work on two ATOMS.');
          writeln(outfile,'ADD must work on two ATOMS.');
          error := true;
          goto 99
        end;  {else}
      end  {if}
      else begin                                        {ERROR-JUMP TO LABEL}
        writeln;writeln(outfile);
        writeln('ADD must work on two elements.');
        writeln(outfile,'ADD must work on two elements.');
        error := true;
        goto 99
      end;  {else}
    end  {if}
    else begin                                          {DO NOT HAVE A LIST}
      writeln;writeln(outfile);
      writeln('ADD must work on a SEQUENCE.');
      writeln(outfile,'ADD must work on a SEQUENCE.');
      error := true;
      goto 99
    end;  {else}
```

```
end;  {ADD}
```

```
{*************************************************************************}
{*************************************************************************}

function ATOM (x:ptr) : ptr;


{**C**  THIS PRIMITIVE FUNCTION TAKES A POINTER TO AN OBJECT, DETER-
       MINES WHETHER THE OBJECT IS AN ATOM, CREATES A DATA REPRE-
       SENTATION FOR THE BOOLEAN RESULT, AND PASSES A POINTER TO
       THIS STRUCTURE.}

{*************************************************************************}
{*************************************************************************}

  var result : boolean;
  begin
    if x^.nodetag = 0 then                          {ATOM}
      result := true
    else                                          {SEQUENCE}
      result := false;
    new(x);                                      {CREATE A NODE}
    x^.backptr := nil;
    x^.link := nil;
    x^.nodetag := 0;
    x^.datatag := 2;
    x^.bool := result;
    ATOM := x
  end;  {ATOM}
```

```
{**************************************************************************}
{**************************************************************************}

function DIV (x:ptr) : ptr;


{**C**  THIS PRIMITIVE FUNCTION TAKES A POINTER TO A LIST WITH TWO
       ELEMENTS, DIVIDES THE TWO ELEMENTS, CREATES A DATA REPRESEN-
       TATION FOR THE RESULT, AND PASSES A POINTER TO THIS STRUCTURE.}

{**************************************************************************}
{**************************************************************************}

  var realsum : real;
  begin
    if x^.nodetag = 1 then begin                {SEQUENCE}
      if (x^.numelements = 2) then begin     {EXACTLY 2 ELEMENTS}
        if (x^.sublist^.nodetag = 0) and
           (x^.sublist^.link^.nodetag = 0) then begin  {BOTH ARE ATOMS}
          if (x^.sublist^.datatag = 0) and
             (x^.sublist^.link^.datatag = 0) then      {BOTH ARE REALS}
            if (x^.sublist^.link^.num <> 0) then begin
              realsum := x^.sublist^.num / x^.sublist^.link^.num;
              new(x);
              x^.backptr := nil;
              x^.link := nil;
              x^.nodetag := 0;
              x^.datatag := 0;
              x^.num := realsum;
              DIV := x
            end  {if}
            else begin                    {ATTEMPTING DIVISION BY 0}
              writeln;writeln(outfile);
              writeln('Attempting division by 0.');
              writeln(outfile,'Attempting division by 0.');
              error := true;
              goto 99
```

```
            end  {else}
          else begin                                  {ERROR-JUMP TO LABEL}
            writeln;writeln(outfile);
            writeln('DIV must work on reals.');
            writeln(outfile,'DIV must work on reals.');
            error := true;
            goto 99
          end;  {else}
        end  {if}
        else begin                                    {ERROR-JUMP TO LABEL}
          writeln;writeln(outfile);
          writeln('DIV must work on two ATOMS.');
          writeln(outfile,'DIV must work on two ATOMS.');
          error := true;
          goto 99
        end;  {else}
      end  {if}
      else begin                                      {ERROR-JUMP TO LABEL}
        writeln;writeln(outfile);
        writeln('DIV must w/rk on two elements.');
        writeln(outfile,'DIV must work on two elements.');
        error := true;
        goto 99
      end;  {else}
    end  {if}
    else begin                                        {DO NOT HAVE A LIST}
      writeln;writeln(outfile);
      writeln('DIV must work on a SEQUENCE.');
      writeln(outfile,'DIV must work on a SEQUENCE.');
      error := true;
      goto 99
    end;  {else}
  end;  {DIV}
```

```
{*************************************************************************}
{*************************************************************************}

function EQUALS (x:ptr) : ptr;


{**C**  THIS PRIMITIVE FUNCTION TAKES A POINTER TO AN OBJECT, DE-
       TERMINES IF ITS TWO ELEMENTS ARE IDENTICAL, CREATES A DATA
       REPRESENTATION FOR THE BOOLEAN RESULT, AND PASSES A POINTER
       TO THIS STRUCTURE.}

{*************************************************************************}
{*************************************************************************}

  var  first,last       :  ptr;
       result           :  boolean;


{*************************************************************************}

function equals (p,q : ptr) : boolean;

{**C**  THIS FUNCTION RECURSIVELY CHECKS THE ELEMENTS OF TWO LISTS
       POINTED TO BY P AND Q AND RETURNS THE BOOLEAN RESULT.}

{*************************************************************************}

  var result : boolean;
  begin
    result := true;                              {ELEMENTS ARE THE SAME}
    p := p^.sublist;
    q := q^.sublist;                             {POINT TO NEXT ELEMENT}
    while (p <> nil) and (q <> nil) and result do begin
      if p^.nodetag = q^.nodetag then begin                {SAME TYPE}
        if p^.nodetag = 0 then begin                         {ATOM}
          if p^.datatag = q^.datatag then begin   {SAME TYPE OF DATA}
            case p^.datatag of
```

```
                0:  if p^.num <> q^.num then                    {REAL}
                      result := false;
                1:  if p^.ident <> q^.ident then               {IDENTIFIER}
                      result := false;
                2:  if p^.bool <> q^.bool then                  {BOOLEAN}
                      result := false;
                3:  if p^.int <> q^.int then                    {INTEGER}
                      result := false
              end;  {case}
            end  {if}
            else result := false;
          end    {if}
          else                                                    {LISTS}
            if (p^.numelements = q^.numelements) and  {SAME # ELEMENTS}
               (p^.datatype = q^.datatype) then       {SAME TYPE DATA}
              result := result and equals(p,q)
            else result := false                {# OR DATA TYPE DIFFERENT}
        end  {if}
        else result := false;                {DIFFERENT KINDS OF ELEMENTS}
        p := p^.link;
        q := q^.link
      end;  {while}
      if (p <> q) then result := false;
      equals := result
    end;  {equals}
```

```
begin                                                    {**EQUALS**}
  if x^.nodetag = 1 then begin                            {SEQUENCE}
    if x^.numelements = 2 then begin                     {TWO-ELEMENT LIST}
      result := true;
      first := x^.sublist;
      last := first^.link;
      if first^.nodetag = last^.nodetag then begin
        if first^.nodetag = 0 then begin                          {ATOMS}
          if first^.datatag = last^.datatag then begin {SAME DATA}
            case first^.datatag of
              0:  if first^.num <> last^.num then          {REAL}
                    result := false;
              1:  if first^.ident <> last^.ident then    {IDENTIFIER}
                    result := false;
              2:  if first^.bool <> last^.bool then       {BOOLEAN}
                    result := false;
              3:  if first^.int <> last^.int then         {INTEGER}
                    result := false
            end;  {case}
          end  {if}
          else result := false              {DIFFERENT KINDS OF DATA}
        end  {if}
        else begin                                          {SEQUENCE}
          if (first^.numelements = last^.numelements) and {SAME #}
             (first^.datatype = last^.datatype) then       {SAME TYPE}
            result := equals(first,last) {RECURSIVELY CHECK ELEMENTS}
          else result := false                           {NOT THE SAME}
        end;  {else}
      end  {if}
      else result := false;              {DIFFERENT KINDS OF ELEMENTS}
    end  {if}
    else begin                           {MO      .  _  _EMENTS IN LIST}
      writeln;writeln(outfile);
      writeln('EQUALS must work on a 2-element List.');
      writeln(outfile,'EQUALS must work on a 2-element List.');
      error := true;
      goto 99
```

```
      end;  {else}
    end  {else}
    else begin                                          {ATOM OR UNDEFINED}
      writeln;writeln(outfile);
      writeln('EQUALS must work on a SEQUENCE.');
      writeln(outfile,'EQUALS must work on a SEQUENCE.');
      error := true;
      goto 99
    end;  {else}
    new(x);                                              {CREATE A NODE}
    x^.backptr := nil;
    x^.link := nil;
    x^.nodetag := 0;
    x^.datatag := 2;
    x^.bool := result;
    EQUALS := x
  end;  {EQUALS}
```

```
{*************************************************************************}
{*************************************************************************}

function GETALFA (x:ptr) : alfa;


{**C**  THIS FUNCTION TAKES A POINTER TO A LIST WITH AN ALFA VALUE
       IN THE DATA FIELD AND RETURNS THE ALFA VALUE.}

{*************************************************************************}
{*************************************************************************}

  begin
    if x <> nil then begin                          {NODE PRESENT}
      if x^.nodetag = 0 then begin                   {DATA NODE}
        if x^.datatag = 1 then                      {ALFA VALUE}
          GETALFA := x^.ident
        else begin                                {ERROR-JUMP TO LABEL}
          writeln;writeln(outfile);
          writeln('GETALFA cannot get a non-alfa value.');
          writeln(outfile,'GETALFA cannot get a non-alfa value.');
          error := true;
          goto 99
        end;  {else}
      end  {if}
      else begin                                  {ERROR-JUMP TO LABEL}
        writeln;writeln(outfile);
        writeln('GETALFA cannot work on a SEQUENCE.');
        writeln(outfile,'GETALFA cannot work on a SEQUENCE.');
        error := true;
        goto 99
      end;  {else}
    end  {if}
    else begin                                    {ERROR-JUMP TO LABEL}
      writeln;writeln(outfile);
      writeln('GETALFA reading thru a nil pointer.');
      writeln(outfile,'GETALFA reading thru a nil pointer.');
```

```
      error := true;
      goto 99
    end;  {else}
  end;  {GETALFA}
```

```
{*************************************************************************}
{*************************************************************************}

function GETBOOL (x:ptr) : boolean;


{**C**  THIS FUNCTION TAKES A POINTER TO A LIST WITH A BOOLEAN VALUE
        IN THE DATA FIELD AND RETURNS THE BOOLEAN VALUE.}

{*************************************************************************}
{*************************************************************************}

  begin
    if x <> nil then begin                          {NODE PRESENT}
      if x^.nodetag = 0 then begin                   {DATA NODE}
        if x^.datatag = 2 then                      {BOOLEAN VALUE}
          GETBOOL := x^.bool
        else begin                                {ERROR-JUMP TO LABEL}
          writeln;writeln(outfile);
          writeln('GETBOOL cannot get a non-boolean value.');
          writeln(outfile,'GETBOOL cannot get a non-boolean value.');
          error := true;
          goto 99
        end;  {else}
      end  {if}
      else begin                                  {ERROR-JUMP TO LABEL}
        writeln;writeln(outfile);
        writeln('GETBOOL cannot work on a SEQUENCE.');
        writeln(outfile,'GETBOOL cannot work on a SEQUENCE.');
        error := true;
        goto 99
      end;  {else}
    end  {if}
    else begin                                    {ERROR-JUMP TO LABEL}
      writeln;writeln(outfile);
      writeln('GETBOOL reading thru a nil pointer.');
      writeln(outfile,'GETBOOL reading thru a nil pointer.');
```

```
      error := true;
      goto 99
    end;  {else}
  end;  {GETBOOL}
```

```
{*************************************************************************}
{*************************************************************************}

function GETINT (x:ptr) : integer;


{**C**  THIS FUNCTION TAKES A POINTER TO A LIST WITH AN INTEGER VALUE
       IN THE DATA FIELD AND RETURNS THE INTEGER VALUE.}

{*************************************************************************}
{*************************************************************************}

  begin
    if x <> nil then begin                          {NODE PRESENT}
      if x^.nodetag = 0 then begin                   {DATA NODE}
        if x^.datatag = 3 then                      {INTEGER VALUE}
          GETINT := x^.int
        else begin                                {ERROR-JUMP TO LABEL}
          writeln;writeln(outfile);
          writeln('GETINT cannot get a non-integer value.');
          writeln(outfile,'GETINT cannot get a non-integer value.');
          error := true;
          goto 99
        end;  {else}
      end  {if}
      else begin                                  {ERROR-JUMP TO LABEL}
        writeln;writeln(outfile);
        writeln('GETINT cannot work on a SEQUENCE.');
        writeln(outfile,'GETINT cannot work on a SEQUENCE.');
        error := true;
        goto 99
      end;  {else}
    end  {if}
    else begin                                    {ERROR-JUMP TO LABEL}
      writeln;writeln(outfile);
      writeln('GETINT reading thru a nil pointer.');
      writeln(outfile,'GETINT reading thru a nil pointer.');
```

```
      error := true;
      goto 99
    end;  {else}
  end;  {GETINT}
```

```
{*************************************************************************}
{*************************************************************************}

function GETREAL (x:ptr) : real;


{**C**  THIS FUNCTION TAKES A POINTER TO A LIST WITH A REAL VALUE
        IN THE DATA FIELD AND RETURNS THE REAL VALUE.}

{*************************************************************************}
{*************************************************************************}

  begin
    if x <> nil then begin                          {NODE PRESENT}
      if x^.nodetag = 0 then begin                   {DATA NODE}
        if x^.datatag = 0 then                      {REAL VALUE}
          GETREAL := x^.num
        else begin                                {ERROR-JUMP TO LABEL}
          writeln;writeln(outfile);
          writeln('GETREAL cannot get a non-real value.');
          writeln(outfile,'GETREAL cannot get a non-real value.');
          error := true;
          goto 99
        end;  {else}
      end  {if}
      else begin                                  {ERROR-JUMP TO LABEL}
        writeln;writeln(outfile);
        writeln('GETREAL cannot work on a SEQUENCE.');
        writeln(outfile,'GETREAL cannot work on a SEQUENCE.');
        error := true;
        goto 99
      end;  {else}
    end  {if}
    else begin                                    {ERROR-JUMP TO LABEL}
      writeln;writeln(outfile);
      writeln('GETREAL reading thru a nil pointer.');
      writeln(outfile,'GETREAL reading thru a nil pointer.');
```

```
      error := true;
      goto 99
    end;  {else}
  end;  {GETREAL}
```

```
{*************************************************************************}
{*************************************************************************}

function ID (x:ptr) : ptr;


{**C**  THIS PRIMITIVE FUNCTION RETURNS THE IDENTITY OF AN OBJECT,
        WHICH IS THE OBJECT ITSELF.}

{*************************************************************************}
{*************************************************************************}

  begin
    ID := x;
  end;  {ID}
```

```
{**************************************************************************}
{**************************************************************************}

function LENGTH (x:ptr) : ptr;


{**C**  THIS PRIMITIVE FUNCTION TAKES A POINTER TO A LIST STRUCTURE,
       DETERMINES THE NUMBER OF ELEMENTS IN THE LIST, CREATES A
       DATA REPRESENTATION FOR THE RESULT, AND PASSES A POINTER TO
       THIS REPRESENTATION.}

{**************************************************************************}
{**************************************************************************}

  var result : integer;
  begin
    if (x^.nodetag = 1) then begin                {SEQUENCE}
      result := x^.numelements;
      new(x);                                   {CREATE A NODE}
      x^.backptr := nil;
      x^.link := nil;
      x^.nodetag := 0;
      x^.datatag := 3;
      x^.int := result;
      LENGTH := x
    end  {if}
    else begin                                 {ERROR-JUMP TO FLAG}
      error := true;
      writeln;writeln(outfile);
      writeln('LENGTH must work on a SEQUENCE');
      writeln(outfile,'LENGTH must work on a SEQUENCE');
      goto 99
    end;  {else}
  end;  {LENGTH}
```

```
{*************************************************************************}
{*************************************************************************}

function MUL (x:ptr) : ptr;


{**C**  THIS PRIMITIVE FUNCTION TAKES A POINTER TO A LIST WITH TWO
       ELEMENTS, MULTIPLYS THE TWO ELEMENTS, CREATES A DATA REP-
       RESENTATION FOR THE RESULT, AND PASSES A POINTER TO THIS
       STRUCTURE.}

{*************************************************************************}
{*************************************************************************}

  var realsum  :  real;
      intsum   :  integer;
  begin
    if x^.nodetag = 1 then begin                          "   {SEQUENCE}
      if (x^.numelements = 2) then begin                   {EXACTLY 2 ELEMENTS}
        if (x^.sublist^.nodetag = 0) and
           (x^.sublist^.link^.nodetag = 0) then begin  {BOTH ARE ATOMS}
          if (x^.sublist^.datatag = 0) and
             (x^.sublist^.link^.datatag = 0) then begin {BOTH ARE REALS}
            realsum := x^.sublist^.num * x^.sublist^.link^.num;
            new(x);                                          {CREATE A NODE}
            x^.backptr := nil;
            x^.link := nil;
            x^.nodetag := 0;
            x^.datatag := 0;
            x^.num := realsum;
            MUL := x
          end  {if}
          else if (x^.sublist^.datatag = 3) and
                  (x^.sublist^.link^.datatag = 3) then begin  {INTEGERS}
            intsum := x^.sublist^.int + x^.sublist^.link^.int;
            new(x);                                          {CREATE A NODE}
            x^.backptr := nil;
```

```
            x^.link := nil;
            x^.nodetag := 0;
            x^.datatag := 3;
            x^.int := intsum;
            MUL := x
          end  {if}
          else begin                                    {ERROR-JUMP TO LABEL}
            writeln;writeln(outfile);
            writeln('MUL can only multiply reals or integers.');
            writeln(outfile,'MUL can only multiply reals or integers.');
            error := true;
            goto 99
          end;  {else}
        end  {if}
        else begin                                      {ERROR-JUMP TO LABEL}
          writeln;writeln(outfile);
          writeln('MUL must work on two ATOMS.');
          writeln(outfile,'MUL must work on two ATOMS.');
          error := true;
          goto 99
        end;  {else}
      end  {if}
      else begin                                        {ERROR-JUMP TO LABEL}
        writeln;writeln(outfile);
        writeln('MUL must work on two elements.');
        writeln(outfile,'MUL must work on two elements.');
        error := true;
        goto 99
      end;  {else}
    end  {if}
    else begin                                          {DO NOT HAVE A LIST}
      writeln;writeln(outfile);
      writeln('MUL must work on a SEQUENCE.');
      writeln(outfile,'MUL must work on a SEQUENCE.');
      error := true;
      goto 99
    end;  {else}
```

```
end;  {MUL}
```

```
{*************************************************************************}
{*************************************************************************}

function NULL (x:ptr) : ptr;


{**C**  THIS PRIMITIVE FUNCTION TAKES A POINTER TO AN OBJECT, DETER-
       MINES IF IT CONTAINS ANY ELEMENTS, CREATES A DATA REPRESEN-
       TATION FOR THE BOOLEAN RESULT, AND PASSES A POINTER TO THIS
       STRUCTURE.}

{*************************************************************************}
{*************************************************************************}

  var result : boolean;
  begin
    if (x^.nodetag = 1) then begin              {SEQUENCE}
      if (x^.numelements = 0) then             {LIST IS EMPTY}
        result := true
      else                                     {LIST IS NOT EMPTY}
        result := false;
      new(x);                                  {CREATE A NODE}
      x^.backptr := nil;
      x^.link := nil;
      x^.nodetag := 0;
      x^.datatag := 2;
      x^.bool := result;
      NULL := x
    end  {if}
    else begin                                 {ATOM OR UNDEFINED}
      writeln;writeln(outfile);
      writeln('NULL must work on a SEQUENCE.');
      writeln(outfile,'NULL must work on a SEQUENCE.');
      error := true;
      goto 99
    end;  {else}
  end;  {NULL}
```

```
{*************************************************************************}
{*************************************************************************}

function PUTALFA (x:alfa) : ptr;


{**C**  THIS FUNCTION TAKES AN ALFA VALUE, CREATES A DATA REPRE-
       SENTATION FOR THIS VALUE, AND RETURNS A POINTER TO THIS
       STRUCTURE.}

{*************************************************************************}
{*************************************************************************}

  var y : ptr;
  begin
    new(y);                                        {CREATE A NODE}
    y^.backptr := nil;
    y^.link := nil;
    y^.nodetag := 0;                                   {DATA}
    y^.datatag := 1;                                   {ALFA}
    y^.ident := x;
    PUTALFA := y                                    {SET POINTER}
  end;  {PUTALFA}
```

```
{*************************************************************************}
{*************************************************************************}

function PUTBOOL (x:boolean) : ptr;


{**C**  THIS FUNCTION TAKES A BOOLEAN VALUE, CREATES A DATA REPRE-
       SENTATION FOR THIS VALUE, AND RETURNS A POINTER TO THIS
       STRUCTURE.}

{*************************************************************************}
{*************************************************************************}

  var y : ptr;
  begin
    new(y);                                          {CREATE A NODE}
    y^.backptr := nil;
    y^.link := nil;
    y^.nodetag := 0;                                     {DATA}
    y^.datatag := 2;                                   {BOOLEAN}
    y^.bool := x;
    PUTBOOL := y                                     {SET POINTER}
  end;  {PUTBOOL}
```

```
{*************************************************************************}
{*************************************************************************}

function PUTINT (x:integer) : ptr;


{**C**  THIS FUNCTION TAKES AN INTEGER VALUE, CREATES A DATA REPRE-
       SENTATION FOR THIS VALUE, AND RETURNS A POINTER TO THIS
       STRUCTURE.}

{*************************************************************************}
{*************************************************************************}

  var y : ptr;
  begin
    new(y);                                          {CREATE A NODE}
    y^.backptr := nil;
    y^.link := nil;
    y^.nodetag := 0;                                     {DATA}
    y^.datatag := 3;                                   {INTEGER}
    y^.int := x;
    PUTINT := y                                      {SET POINTER}
  end;  {PUTINT}
```

```
{*************************************************************************}
{*************************************************************************}

function PUTREAL (x:real) : ptr;


{**C**  THIS FUNCTION TAKES A REAL VALUE, CREATES A DATA REPRE-
       SENTATION FOR THIS VALUE, AND RETURNS A POINTER TO THIS
       STRUCTURE.}

{*************************************************************************}
{*************************************************************************}

  var y : ptr;
  begin
    new(y);                                          {CREATE A NODE}
    y^.backptr := nil;
    y^.link := nil;
    y^.nodetag := 0;                                     {DATA}
    y^.datatag := 0;                                     {REAL}
    y^.num := x;
    PUTREAL := y                                       {SET POINTER}
  end;  {PUTREAL}
```

```
{*************************************************************************}
{*************************************************************************}

function REV (x:ptr) : ptr;


{**C**  THIS PRIMITIVE FUNCTION REVERSES THE ELEMENTS OF A LIST.}

{*************************************************************************}
{*************************************************************************}

  var first,final,next,temp  : ptr;
  begin
    if x^.nodetag = 1 then begin                          {SEQUENCE}
      if x^.numelements < 2 then REV := x     {0 OR 1 ELEMENT LIST}
      else begin                                {2 OR MORE ELEMENT LIST}
        next := x^.sublist;
        first := next;
        while next^.link <> nil do              {FIND END OF LIST}
          next := next^.link;
        final := next;
        x^.sublist := final;
        temp := next^.backptr;
        next^.backptr := x;
        next^.link := temp;
        next := next^.link;
        while next <> first do begin    {CONTINUE UNTIL 1ST ELEMENT}
          temp := next^.backptr;
          next^.backptr := next^.link;
          next^.link := temp;
          next := next^.link
        end;  {while}
        next^.backptr := next^.link;
        next^.link := nil;                      {FINAL PTR = NIL}
        REV := x
      end  {else}
    end  {if}
```

```
  else begin                                    {ATOM OR UNDEFINED}
    writeln;writeln(outfile);
    writeln('REV must work on a SEQUENCE.');
    writeln(outfile,'REV must work on a SEQUENCE.');
    error := true;
    goto 99
  end;  {else}
end;  {REV}
```

```
{*************************************************************************}
{*************************************************************************}

function SELECT (x : ptr) : ptr;


{**C**  THIS PRIMITIVE FUNCTION TAKES A POINTER TO A LIST STRUCTURE
       THAT LOOKS LIKE:  ( +n , (a list) ), SELECTS THE nth ELE-
       MENT OF (a list), AND RETURNS A POINTER TO THAT ELEMENT.}

{*************************************************************************}
{*************************************************************************}

  var first,list,move  :  ptr;
      count            :  real;
  begin
    if (x^.nodetag = 1) and (x^.numelements = 2) then begin {2-LIST}
      first := x^.sublist;
      if (first^.nodetag = 0) and                          {ATOM}
         (first^.datatag = 0) and                         {INTEGER}
         (first^.num > 0) then begin                      {POSITIVE}
        list := first^.link;                             {GO TO LIST}
        if (list^.nodetag = 1) and                        {LIST}
           (list^.numelements >= first^.num) then begin {n>=s}
          move := list^.sublist;
          count := 1.0;
          while count < first^.num do begin              {DO s TIMES}
            move := move^.link;
            count := count + 1.0
          end;  {while}
          SELECT := move                                 {SET POINTER}
        end  {if}
        else begin                                      {ERROR-JUMP TO LABEL}
          writeln;writeln(outfile);
          writeln('SELECT must work on a List with n>=s.');
          writeln(outfile,'SELECT must work on a List with n>=s.');
          error := true;
```

```
            goto 99
          end;  {else}
        end  {if}
        else begin                                        {ERROR-JUMP TO LABEL}
          writeln;writeln(outfile);
          writeln('SELECT''s first element must be a positive integer.');
          writeln(outfile,'SELECT''s first element must be a positive integer.');
          error := true;
          goto 99
        end;  {else}
      end  {if}
      else begin                                          {ERROR-JUMP TO LABEL}
        writeln;writeln(outfile);
        writeln('SELECT must work on a 2 element List.');
        writeln(outfile,'SELECT must work on a 2 element List.');
        error := true;
        goto 99
      end;  {else}
    end;  {SELECT}
```

```
{*************************************************************************}
{*************************************************************************}

function SUB (x:ptr) : ptr;


{**C**  THIS PRIMITIVE FUNCTION TAKES A POINTER TO A LIST WITH TWO
       ELEMENTS, SUBTRACTS THE TWO ELEMENTS, CREATES A DATA REP-
       RESENTATION FOR THE RESULT, AND PASSES A POINTER TO THIS
       STRUCTURE.}

{*************************************************************************}
{*************************************************************************}

  var realsum  :  real;
      intsum   :  integer;
  begin
    if x^.nodetag = 1 then begin                          {SEQUENCE}
      if (x^.numelements = 2) then begin                {EXACTLY 2 ELEMENTS}
        if (x^.sublist^.nodetag = 0) and
           (x^.sublist^.link^.nodetag = 0) then begin  {BOTH ARE ATOMS}
          if (x^.sublist^.datatag = 0) and
             (x^.sublist^.link^.datatag = 0) then begin {BOTH ARE REALS}
            realsum := x^.sublist^.num - x^.sublist^.link^.num;
            new(x);                                       {CREATE A NODE}
            x^.backptr := nil;
            x^.link := nil;
            x^.nodetag := 0;
            x^.datatag := 0;
            x^.num := realsum;
            SUB := x
          end  {if}
          else if (x^.sublist^.datatag = 3) and
                  (x^.sublist^.link^.datatag = 3) then begin  {INTEGERS}
            intsum := x^.sublist^.int + x^.sublist^.link^.int;
            new(x);                                       {CREATE A NODE}
            x^.backptr := nil;
```

```
            x^.link := nil;
            x^.nodetag := 0;
            x^.datatag := 3;
            x^.int := intsum;
            SUB := x
          end  {if}
          else begin                                    {ERROR-JUMP TO LABEL}
            writeln;writeln(outfile);
            writeln('SUB can only add reals or integers.');
            writeln(outfile,'SUB can only add reals or integers.');
            error := true;
            goto 99
          end;  {else}
        end  {if}
        else begin                                      {ERROR-JUMP TO LABEL}
          writeln;writeln(outfile);
          writeln('SUB must work on two ATOMS.');
          writeln(outfile,'SUB must work on two ATOMS.');
          error := true;
          goto 99
        end;  {else}
      end  {if}
      else begin                                        {ERROR-JUMP TO LABEL}
        writeln;writeln(outfile);
        writeln('SUB must work on two elements.');
        writeln(outfile,'SUB must work on two elements.');
        error := true;
        goto 99
      end;  {else}
    end  {if}
    else begin                                          {DO NOT HAVE A LIST}
      writeln;writeln(outfile);
      writeln('SUB must work on a SEQUENCE.');
      writeln(outfile,'SUB must work on a SEQUENCE.');
      error := true;
      goto 99
    end;  {else}
```

```
end;  {SUB}
```

```
{*************************************************************************}
{*************************************************************************}

function TAIL (x:ptr) : ptr;

{**C**  THIS PRIMITIVE FUNCTION DISGARDS THE FIRST ELEMENT OF A LIST
       AND RETURNS A LIST CONTAINING THE REST OF THE ELEMENTS OF
       THE LIST.}

{*************************************************************************}
{*************************************************************************}

  var alphaflag    :  boolean;
      nextelement  :  ptr;
  begin
    alphaflag := false;
    if (x^.nodetag = 1) and (x^.numelements > 1) then      {SEQUENCE}
                                                            { n>=2 }
      begin
        x^.sublist := x^.sublist^.link;              {SKIP 1ST ELEMENT}
        x^.sublist^.backptr := x;
        x^.numelements := x^.numelements - 1;        {DELETE 1 ELEMENT}
        nextelement := x^.sublist;
        while nextelement <> nil do begin
          if nextelement^.nodetag = 1 then             {SEQUENCE}
            if nextelement^.datatype = 'A' then
              alphaflag := true
          else                                          {ATOM}
            if nextelement^.datatag = 1 then          {ATOM IS ALPHA}
              alphaflag := true;
          nextelement := nextelement^.link;
        end;  {while}
        if alphaflag then
          x^.datatype := 'A'
        else
          x^.datatype := 'N';
```

```
      TAIL := x;
    end
  else begin                                        {ATOM OR UNDEFINED}
    writeln;writeln(outfile);
    writeln('TAIL must work on a SEQUENCE with > 1 element.');
    writeln(outfile,'TAIL must work on a SEQUENCE with > 1 element.');
    error := true;
    goto 99
  end;  {else}
end;  {TAIL}
```

```
{*************************************************************************}
{*************************************************************************}

function APTALL (function f (x1:ptr) : ptr; x:ptr) : ptr;


{**C**  THIS FUNCTIONAL PERFORMS A SELECTED PRIMITIVE FUNCTION
       OPERATION ON EACH ELEMENT IN A LIST STRUCTURE.}

{*************************************************************************}
{*************************************************************************}

  var first,move,next,y : ptr;
      firstnode         : boolean;
  begin
    if x^.nodetag = 1 then                                  {SEQUENCE}
      begin
        if x^.numelements = 0 then begin                    {EMPTY SET}
          new(y);                                  {CREATE AN EMPTY LIST}
          first := y;
          y^.backptr := nil;
          y^.link := nil;
          y^.nodetag := 1;
          y^.numelements := 0;
          y^.datatype := 'U';
          new(y^.sublist);                                  {CREATE A NODE}
          next := y^.sublist;
          next^.backptr := first;
          next^.link := nil;
          next^.nodetag := 0;
          next^.datatag := 1;
          next^.ident := 'none      ';
          APTALL := y
        end  {if}
        else                                                {NOT EMPTY SET}
          begin
            firstnode := true;
```

```
                next := x^.sublist;
                move := x;
                repeat
                  y := f(next);                    {POINT TO RESULT OF FUNCTION}
                  if firstnode then
                    begin
                      firstnode := false;
                      move^.sublist := y;
                      y^.backptr := move;
                      move := move^.sublist;
                    end
                  else
                    begin
                      y^.backptr := move;
                      move^.link := y;
                      move := move^.link;
                    end;
                  next := next^.link;
                until next = nil;
                APTALL := x
              end;
          end
        else begin                                 {ATOM OR UNDEFINED ENTITY}
          writeln;writeln(outfile);
          writeln('APTALL must work on a SEQUENCE.');
          writeln(outfile,'APTALL must work on a SEQUENCE.');
          error := true;
          goto 99
        end;  {else}
      end;  {APTALL}
```

```
{************************************************************************}
{************************************************************************}

function INSERT (function f (x1:ptr) : ptr; x:ptr) : ptr;


{**C**  THIS FUNCTIONAL PERFORMS A SELECTED BINARY ARITHMETIC
       OPERATION ON AN ENTIRE LIST STRUCTURE BEGINNING WITH THE
       INNERMOST LEVEL (RIGHT TO LEFT) AND USING THE RESULT AS
       INPUT TO THE NEXT HIGHER LEVEL UNTIL THE OUTERMOST
       LEVEL IS REACHED AND THE FINAL RESULT OBTAINED.}

{************************************************************************}
{************************************************************************}

  var alldone          :  boolean;
      move,reverse,y   :  ptr;
      result           :  real;
  begin
    if x^.nodetag = 1 then begin                          {SEQUENCE}
      if x^.numelements = 0 then begin                    {EMPTY SET}
        new(x);                                         {CREATE EMPTY SET}
        x^.backptr := nil;
        x^.link := nil;
        x^.nodetag := 1;
        x^.numelements := 0;
        x^.datatype := 'U';
        new(x^.sublist);
        move := x^.sublist;
        move^.backptr := x;
        move^.link := nil;
        move^.nodetag := 0;
        move^.datatag := 1;
        move^.ident := 'none      ';
        INSERT := x
      end  {if}
      else if (x^.numelements = 1) and (x^.sublist^.nodetag = 0) then
```

```
                                                {ONE ELEMENT IN SEQUENCE}
  begin
    result := x^.sublist^.num;
    new(x);
    x^.backptr := nil;
    x^.link := nil;
    x^.nodetag := 0;
    x^.datatag := 0;
    x^.num := result;
    INSERT := x;
  end
else begin                          {MORE THAN ONE ELEMENT IN SEQUENCE}
  if x^.datatype = 'N' then begin    {ELEMENTS ARE NUMBERS}
    move := x^.sublist;
    repeat
      if move^.nodetag = 0 then begin  {ELEMENTS ARE ATOMS}
        if move^.link = nil then
          reverse := move;
        move := move^.link;
      end  {if}
      else begin                          {ELEMENTS NOT ALL ATOMS}
        writeln;writeln(outfile);
        writeln('The elements of INSERT cannot be SEQUENCES.');
        writeln(outfile,'The elements of INSERT cannot be SEQUENCES.');
        error := true;
        goto 99
      end;  {else}
    until move = nil;
    x^.numelements := 2;
    alldone := false;
    repeat
      x^.sublist := reverse^.backptr;
      reverse := reverse^.backptr;
      y := f(x);
      if reverse^.backptr <> x then begin
        x^.sublist^.backptr^.link := y;
        y^.backptr := x^.sublist^.backptr
```

```
            end  {if}
            else alldone := true;
          until alldone;
          INSERT := y;
        end  {if}
        else begin                                  {ELEMENTS ARE NOT NUMBERS}
          writeln;writeln(outfile);
          writeln('INSERT must work on a SEQUENCE of numbers.');
          writeln(outfile,'INSERT must work on a SEQUENCE of numbers.');
          error := true;
          goto 99
        end;  {else}
      end  {else}
    end  {if}
    else begin                                      {ATOM OR UNDEFINED ENTITY}
      writeln;writeln(outfile);
      writeln('INSERT must work on a SEQUENCE.');
      writeln(outfile,'INSERT must work on a SEQUENCE.');
      error := true;
      goto 99
    end;  {else}
  end;  {INSERT}
```

```
{*************************************************************************}
{*************************************************************************}

function TRANS (x:ptr) : ptr;


{**C**  THIS FUNCTIONAL TRANSPOSES A  m by n  LIST (where m is the
       number of elements in the List, and n is the number of
       elements in that element) INTO A  n by m  LIST, AND PASSES
       A POINTER TO THIS STRUCTURE.}

{*************************************************************************}
{*************************************************************************}

  var firstdata,firstnode          :  boolean;
      total,subtotal,i             :  integer;
      first,move,previous,y,z      :  ptr;

{*************************************************************************}

procedure COPY;

{**C**  THIS PROCEDURE CREATES THE NEW DATA REPRESENTATION OF THE
       TRANSPOSED LIST.}

{*************************************************************************}

  begin
    if firstdata then begin                    {FIRST ELEMENT IN SUB-LIST}
      new(z^.sublist);                           {CREATE A NODE}
      z^.sublist^.backptr := z;
      z^.sublist^.nodetag := move^.sublist^.nodetag;
      case z^.sublist^.nodetag of
        0:  begin                                          {ATOM}
```

```
            z^.sublist^.datatag := move^.sublist^.datatag;
            case z^.sublist^.datatag of
              0:  z^.sublist^.num := move^.sublist^.num;  {REAL}
              1:  z^.sublist^.ident := move^.sublist^.ident;{ALFA}
              2:  z^.sublist^.bool := move^.sublist^.bool; {BOOLEAN}
              3:  z^.sublist^.int := move^.sublist^.int    {INTEGER}
            end;  {case}
          end;  {begin}
        1:  begin                                          {SEQUENCE}
            z^.sublist^.numelements := move^.sublist^.numelements;
            z^.sublist^.datatype := move^.sublist^.datatype;
            z^.sublist^.sublist := move^.sublist^.sublist
          end  {begin}
      end;  {case}
      z := z^.sublist;
      firstdata := false
    end  {if}
    else begin                              {NOT 1ST ELEMENT IN SUB-LIST}
      case move^.sublist^.nodetag of
        0:  begin                                             {ATOM}
            new(z^.link);                                  {CREATE A NODE}
            z^.link^.backptr := z;
            z^.link^.nodetag := move^.sublist^.nodetag;
            z^.link^.datatag := move^.sublist^.datatag;
            case z^.link^.datatag of
              0:  z^.link^.num := move^.sublist^.num;       {REAL}
              1:  z^.link^.ident := move^.sublist^.ident;   {ALFA}
              2:  z^.link^.bool := move^.sublist^.bool;    {BOOLEAN}
              3:  z^.link^.int := move^.sublist^.int       {INTEGER}
            end;  {case}
          end;  {begin}
        1:  begin                                          {SEQUENCE}
            new(z^.link);
            z^.link^.backptr := z;
            z^.link^.nodetag := move^.sublist^.nodetag;
            z^.link^.numelements := move^.sublist^.numelements;
            z^.link^.datatype := move^.sublist^.datatype;
```

```
                z^.link^.sublist := move^.sublist^.sublist
              end  {begin}
        end;  {case}
        z := z^.link
      end;  {else}
    end;  {COPY}
```

```
begin                                                    {**TRANS**}
  if x^.nodetag = 1 then begin                            {SEQUENCE}
    if x^.numelements > 1 then begin                  {AT LEAST 2 ELEMENTS}
      move := x^.sublist;
      total := x^.numelements;
      subtotal := move^.numelements;
      while (move <> nil) do begin {CHECK ELEMENTS FOR SAME CONST'N}
        if (move^.nodetag = 0) or
           (move^.numelements <> subtotal) then begin
          writeln;writeln(outfile);
          write('TRANS must work on identically constructed ');
          writeln('elements.');
          write(outfile,'TRANS must work on identically ');
          writeln(outfile,'constructed elements.');
          error := true;
          goto 99
        end;  {if}
        move := move^.link
      end;  {while}
      new(y);                                          {CREATE A NODE}
      y^.backptr := nil;
      y^.link := nil;
      y^.nodetag := 1;
      y^.numelements := subtotal;
      y^.datatype := x^.datatype;
      firstnode := true;
      for i := 1 to subtotal do begin     {CREATE A TRANSPOSED LIST}
        move := x^.sublist;              {POINT BACK TO 1ST ELEMENT}
        new(z);                                {CREAT A NODE}
        z^.nodetag := 1;
        z^.numelements := total;
        if firstnode then begin                  {1ST ELEMENT}
          first := z;
          previous := first;
          firstnode := false;
          y^.sublist := first;
          first^.backptr := y
```

```
            end  {if}
            else begin                                  {NOT 1ST ELEMENT}
              previous^.link := z;
              z^.backptr := previous;
              previous := previous^.link
            end;  {else}
            firstdata := true;
            while (move <> nil) do begin
              COPY;
              move^.sublist := move^.sublist^.link;
              if move^.sublist <> nil then
                move^.sublist^.backptr := move;
              move := move^.link
            end;  {while}
            z^.link := nil
          end;  {for}
        end  {if}
        else begin                                  {NOT ENOUGH ELEMENTS}
          writeln;writeln(outfile);
          write('TRANS must work on a SEQUENCE with at least ');
          writeln('2 elements.');
          write(outfile,'TRANS must work on a SEQUENCE with at least ');
          writeln(outfile,'2 elements.');
          error := true;
          goto 99
        end;  {else}
      end  {if}
      else begin                                    {ATOM}
        writeln;writeln(outfile);
        writeln('TRANS must work on a SEQUENCE.');
        writeln(outfile,'TRANS must work on a SEQUENCE.');
        error := true;
        goto 99
      end;  {else}
      TRANS := y
    end;  {TRANS}
```

APPENDIX B -- USER'S MANUAL

I. INTRODUCTION

Welcome to the Functional PASCAL Programming System. This system is designed to allow the user to write computer programs in either the traditional "conventional" mode or in a "functional" mode. A basic knowledge of applicative programming techniques and the PASCAL programming language is assumed in order to effectively use and understand the Functional PASCAL System. Readers desiring further information on functional programming are referred to the reference list following Appendix C of this thesis.

The User's Manual is divided into six additional sections as follows:

- II. OVERALL PROGRAM FORMAT
- III. INPUT PROCEDURES
- IV. OUTPUT PROCEDURES
- V. SYSTEM-DEFINED FUNCTIONS
- VI. USER-DEFINED FUNCTIONS
- VII. ERROR HANDLING PROCEDURES

II. CVERALL PROGRAM FORMAT

The format of a Functional PASCAL Program is very similar to that of a conventional PASCAL program. The only major difference is that the functional system requires a few additional system files which enables the user to input "objects" for processing by the system and user-defined functions. These functions are provided in the system library and are incorporated into the main program by using the standard "include" statement. In order for the user to gain full access to all of the functional programming capabilities of the system, it is necessary to "include" six components in the main program: (1) a label declaration, (2) a type declaration, (3) a variable declaration, (4) system-defined functions, (5) an initialization routine and (6) an errorlabel declaration. These components provide a complete interface package with the functional portion of the system. The label and errorlabel declarations allow the system to terminate gracefully from the functional mode of operation upon encountering an error condition in that program module. It should be be noted that the "include" statement must follow the format of the sample program shown in this Appendix. The # must appear in column 1 and the quotation marks must surround the designated system files being called.

FUNCTIONAL PASCAL PROGRAM FORMAT

```
PROGRAM funcpascal (input,output);
# include label
  TYPE
#   include "type.i"
    .
    .
    .
    user-defined type declarations
  VAR
#   include "var.i"
    .
    .
    .
    user-defined variable declarations
# include "functions.i"
  .
  .
  .
  user-defined procedures and functions
  BEGIN
#   include "init.i"
    .
    .
    .
    main body of program
    .
    .
    .
#   include "errorlabel.i"
  END.  { program funcpascal }
```

III. INPUT PROCEDURES

The Functional PASCAL System allows the user to enter data into the program via an interactive input function, READLIST. The data is entered upon seeing a prompt to the terminal as follows:

```
ENTER OBJECT.
===>
```

The interactive function, READLIST, can be activated by invoking it as a routine function call as in any other PASCAL program. The following example illustrates a typical statement in the main body of the Functional PASCAL program which accomplishes this task:

```
x := READLIST;
```

The primary purpose of this function is to capture raw data from the user and format it into an appropriate data representation. The return value from READLIST is a pointer variable to the newly-created data structure.

The basic unit of any input data is the "object". An "object" can be an atom or a sequence, depending on the type of functional operation being invoked. The Functional PASCAL System requires the user to follow a limited number of "syntax" rules when entering data objects into a program.

- The allowable character set for data objects is:

(1) A..Z
(2) a..z
(3) Ø..9
(4) -
(5) .

- Atoms can be constructed from a combination of one to ten characters from the allowable character set.

  Identifiers: (1) Must begin with A..Z, a..z
  (2) Can have numbers after first letter
  (3) Not to exceed ten characters

  Numbers: (1) Cannot exceed PASCAL compiler length or will go into overflow condition
  (2) Can input negative numbers
  (3) Can input decimal numbers

- Sequences can be constructed from Ø..n elements where an element is defined to be an atom or another sequence.

- All sequences must begin with an open parenthesis and end with a closed parenthesis.

- The length of an input data object cannot exceed 8Ø characters due to the limitation of the input buffer.

- Every element in a sequence must be separated by a comma.

- Spaces may be inserted at any point in the "object" but are disregarded during formation of the data structure.

Examples of input objects to a Functional PASCAL program:

```
===>cat                 (atom)
===>()                  (sequence)
===>12                  (atom)
===>((1,2),(3,4))       (sequence)
===>(a,b,c,(1,(a)))     (sequence)
===>1.ØE                (atom)
===>(cat,dog,ball)      (sequence)
```

Data input into the system which violates any of the basic syntax rules results in an error message to the user.

IV. OUTPUT PROCEDURES

The Functional PASCAL System provides the user with a comprehensive set of output functions capable of extracting data from the appropriate representation and presenting it in a useable form in the main program. The output functions are divided into two basic catagories: (1) output of sequences and (2) output of atoms.

In order to extract sequence data from the data structure, the procedure WRITELIST is used. This procedure removes the data from the list structure and inserts the "syntactic sugar" (parentheses and commas) before displaying to the terminal or to a file. The input parameter to this procedure is a pointer variable (x) to the current data structure created by the processing of an input data object.

```
WRITELIST(x);      ===>  ((1,2,3),(4,5),(a,b,c))
```

In order to extract non-sequence data from the data structure, a series or GET functions can be invoked, depending on the type of atom that is to be removed. Atoms can be of the form real, alfa, boolean or integer and are contained in a record structure similar to a sequence. The GET function allows the user to access the particular data structure containing the result of a functional operation and make an appropriate assignment to a variable of like type in the main program.

Examples of non-sequence output functions --

y := GETREAL(x);
where x is a pointer to a record containing a real number and y is a variable of type real declared in the main program.

y := GETALFA(x);
where x is a pointer to a record containing an alfa character(s) and y is a variable of type alfa declared in the main program.

y := GETBOOL(x);
where x is a pointer to a record containing a boolean expression and y is a variable of type boolean declared in the main program.

y := GETINT(x);
where x is a pointer to a record containing an integer value and y is a variable of type integer declared in the main program.

A corresponding PUT function is provided for each of the GET operations above. The PUT operation is the complement of the GET operation and takes the raw atom in the form of a real, alfa, boolean or integer and inserts it into a list structure. A pointer variable is returned to the main program after the PUT operation is completed. The GET and PUT operations provide the necessary interface from the functional to the conventional mode and vice versa.

It should be noted that REALLIST converts integer numbers into reals, and therefore, all corresponding outputs, with the exception of LENGTH, appear as real numbers.

V. SYSTEM-DEFINED FUNCTIONS

The Functional PASCAL System (Phase I) provides the user with a total of thirteen primitive functions and three functional forms. These subroutines can be called from the system library at any time following the same rules for invoking procedures and functions as in a conventional PASCAL program. A complete description of input and output parameters for each primitive and functional can be found in Chapter IV of the research and development portion of this thesis.

PRIMITIVE FUNCTIONS

Primitive	Calling Form
1) Select	SELECT
2) Identity	ID
3) Reverse	REV
4) Transpose	TRANS
5) Atom	ATOM
6) Null	NULL
7) Equals	EQUALS
8) Tail	TAIL
9) Length	LENGTH
10) Add	ADD
11) Subtract	SUB
12) Multiply	MUL
13) Divide	DIV

FUNCTIONAL FORMS

Functional	Calling Form
1) Insert	INSERT
2) Apply to all	APTALL
3) Composition	N/A

Each primitive function is standardized in that it takes, as its argument, a pointer variable (x) to the data representation of the input object, performs the appropriate operation on the object and returns a pointer variable to the newly-created data structure after the operation is complete.

Examples of primitive function operations within a Functional PASCAL program --

```
y := ADD(x);
     where x = (6,8.5)
     ===> 14.5

y := SELECT(x);
     where x = (2,(4,6,10,cat,(1,2)))
     ===> 6

y := REV(x);
     where x = ((1,2),ball,5)
     ===> (5,ball,(1,2))

y := TRANS(x);
     where x = ((1,cat,(1,2)),(a,b,c))
     ===> ((1,a),(cat,b),((1,2),c))

     where y is a variable of type "pointer" declared
     in the user's main program.
```

Each of the functional forms is also standardized in that it takes, as its arguments, (1) a system or user-defined function (f) and (2) a pointer variable (x). It then performs the appropriate operation on the object and returns a pointer variable to the newly-created data structure containing the results of the operation.

Examples of functional operations within a Functional PASCAL program --

```
y := APTALL(ADD,x);
     where x = ((1,2),(3,4),(5,6))
     ===> (3,7,11)
```

```
y := INSERT(SUB,x);
     where x = (10,4,6.5,8)
     ===> 4.5

y := APTALL(LENGTH,x);
     where x = ((1,2),(a,b,c,d),(),(3))
     ===> (2,4,0,1)

y := APTALL(TAIL,x);
     where x = ((a,b,c),(cat,dog),(1,(3,4)))
     ===> ((b,c),(dog),((3,4)))

     where y is a variable of type "pointer" declared
     in the user's main program.
```

A third functional form, COMPOSITION, is provided inherently as part of the operating procedure within the UC Berkeley PASCAL compiler. The functional form of COMPOSITION allows the "stacking" of multiple functions in order to provide flexibility in performing more complex operations. The standard format for a COMPOSITION operation is as follows:

$$(f \circ g) : x \;\Longrightarrow\; f : (g : x)$$

where *f* and *g* are system or user-defined functions and x is an input object. The above notation reads "the composition of f and g applied to x".

Examples of the functional operation of COMPOSITION within a Functional PASCAL program --

```
y := REV(TAIL(APTALL(ADD,x)));
     where x = ((1,2),(5,6),(3,5))
     ===> (8,11)

y := INSERT(ADD,APTALL(MUL,x));
     where x = ((3,5),(6,4),(6,8),(1,3))
     ===> 90

y := INSERT(ADD,APTALL(MUL,TRANS(x)))
     where x = ((1,2,3),(6,5,4))
     ===> 28
```

```
y := INSERT(ADD,APTALL(LENGTH,x))
     where x = ((1,2,3),(a,b,c,d),(2,5),(7))
     ===> 10
```

where y is a variable of type "pointer" declared in the user's main program.

VI. USER-DEFINED FUNCTIONS

The Functional PASCAL System provides a basic set of primitives and functionals which can be extended through the use of user-defined functions. A user-defined function is declared in the main program in the same manner as any routine PASCAL function. The function is comprised of more than one primitive or functional and performs a new operation that is beyond the scope of the basic system-defined functions. The user-defined functions follow the same format as the primitive functions inasmuch as a pointer variable is taken as an input parameter and also returned as an output parameter.

Examples of user-defined functions within the Functional PASCAL System --

```
y := INNERPROD(x);
```

where INNERPROD is a function declared in the main program.

```
function INNERPROD (x:ptr) : ptr;
  begin
    INNERPROD := INSERT(ADD,APTALL(MUL,TRANS(x)));
  end;  { function INNERPROD }
```

```
y := COUNTELEMENTS(x);
```

where COUNTELEMENTS is a function declared in the main program.

```
function COUNTELEMENTS (x:ptr) : ptr;
  begin
    COUNTELEMENTS := INSERT(ADD,APTALL(LENGTH,x));
  end;  { function COUNTELEMENTS }
```

VII. ERROR HANDLING PROCEDURES

The Functional PASCAL System provides the user with comprehensive error checking routines. Error handling is accomplished in two distinct phases: (1) during the interactive input function, READLIST and (2) during the actual application of the system and user-defined functions on the input data object.

During the first phase of the error checking routines, the input data object is examined for syntactic errors and appropriate messages are displayed to the terminal. The second phase consists of a semantic evaluation of the functions as applied to the particular input object and, again, display of messages for user information. In both cases, the errors are fatal and execution of the Functional Pascal program is terminated. Other errors outside of the functional portion of the program are handled routinely by the UC Berkeley PASCAL compiler.

APPENDIX C -- INITIAL TEST RESULTS

SECTION I. UNIT TESTING (Primitive Functions)

The following test problems and results are provided to illustrate the initial unit testing conducted on the primitive functions within the Functional PASCAL System.

```
Primitive Function: ADD
Input Object: (10,15.2)
Result ===> 25.20

PF: SUB
IO: ((1,2),3)
RES: Error

PF: MUL
IO: (4,12)
RES: 48.00

PF: DIV
IO: (6,0)
RES: Error

PF: DIV
IO: (0,8)
RES: 0.00

PF: ATOM
IO: ((2))
RES: false

PF: ATOM
IO: cat
RES: true

PF: Null
IO: (a,b,c)
RES: false

PF: NULL
IO: ( )
RES: true

PF: ID
IO: ((1,2,3),(4,5,6))
RES: ((1,2,3),(4,5,6))
```

```
Primitive Function: REV
Input Object: (a,b,c,d,e)
Result ===> (e,d,c,b,a)

PF: REV
IO: (1,(2,4),6,(7,8,9))
RES: ((7,8,9),6,(2,4),1)

PF: LENGTH
IO: (A,B,3,cat,6,4)
RES: 6

PF: LENGTH
IO: (4,3.6,x,y,((1,2)))
RES: 5

PF: EQUALS
IO: (((1,2,(3))),((1,2,(3))))
RES: true

PF: EQUALS
IO: 36
RES: Error

PF: EQUALS
IO: (a,(a))
RES: false

PF: TAIL
IO: (1,2,3,4.5,X)
RES: (2,3,4.5,X)

PF: TAIL
IO: ((1,2,3,(4)),X)
RES: (X)

PF: TAIL
IO: ( )
RES: Error
```

```
PF: SELECT
IO: (5,(X,(1,2),ball,2,10))
RES: 10

PF: SELECT
IO: (1)
RES: Error

PF: TRANS
IO: ((1,2,3),(4,5),(7,8,9))
RES: Error

PF: TRANS
IO: ((1,2),(cat,dog))
RES: ((1,cat),(2,dog))

PF: ADD
IO: (3,5,7)
RES: Error

PF: DIV
IO: (100.6,2.0)
RES: 50.30

PF: NULL
IO: ball
RES: Error

PF: ID
IO: 1
RES: Error

PF: REV
IO: (1)
RES: (1)

PF: SELECT
IO: (4,(x,y,z))
RES: Error

PF: TAIL
IO: a
RES: Error

PF: TRANS
IO: ((1,2,3),(4,5,6))
RES: ((1,4),(2,5),(3,6))

PF: EQUALS
IO: (( ),( ))
RES: true

PF: ADD
IO: 3,4
RES: Error

PF: MUL
IO: (3,0)
RES: 0.00

PF: SUB
IO: (0,8.3)
RES: -8.30

PF: SELECT
IO: 3,a,b,c
RES: Error

PF: ID
IO: a,b,4
RES: Error

PF: REV
IO: ( )
RES: ( )

PF: ATOM
IO: (ball)
RES: false
```

SECTION III. INTEGRATION TESTING (User-defined functions)

The following test problems and results are provided to illustrate the initial integration testing conducted on selected user-defined functions.

****

```
function INNERPROD (x:ptr) : ptr;
  begin
    INNERPROD := INSERT(ADD,APTALL(MUL,TRANS(x)));
  end;  { function INNERPROD }
```

User-defined function: INNERPROD
Purpose: To find the inner product of a matrix.
Input object: ((1,2,3),(6,5,4))
Result: 28

****

```
function BATAVE (x:ptr) : ptr;
  begin
    BATAVE := APTALL(DIV,TRANS(x));
  end;  { function BATAVE }
```

User-defined function: BATAVE
Purpose: To find the batting averages of baseball players given a two-element list (TOTAL HITS and TOTAL AT BATS).
Input object: ((10,15,20,11),(30,60,40,33))
Result: (0.33, 0.25, 0.50, 0.33)

****

```
function TOTLISTELEMENTS (x:ptr) : ptr;
  begin
    TOTLISTELEMENTS := INSERT(ADD,APTALL(LENGTH,x));
  end;  { function TOTLISTELEMENTS }
```

User-defined function: TOTLISTELEMENTS
Purpose: To find the total number of elements in a list of lists.
Input object: ((1,2,3),( ),(a,b,c,d),(cat,dog))
Result: 9

****

SECTION IV. INTEGRATION TESTING (Functional PASCAL program)

The following Functional PASCAL program is provided to illustrate the diverse capabilities of the system in a combination conventional and functional environment.

```
**************************************************************

program baseballstats(input,output,outfile);

# include "label.i"

  const
    limit = 500;

  type
#   include "type.i"

  var
#   include "var.i"
    x : ptr;
    totplayers : integer;

# include "functions.i"

  function BATTINGAVERAGE (x:ptr) : ptr;
    begin
      BATTINGAVERAGE := APTALL(DIV,TRANS(x));
    end;  { function BATTINGAVERAGE }

  function TOTLEAGUEPLAYERS (x:ptr) : ptr;
    begin
      TOTLEAGUEPLAYERS := INSERT(ADD,APTALL(LENGTH,x));
    end;  { function TOTLEAGUEPLAYERS }

  begin
#   include "init.i"
    x := BATTINGAVERAGE (READLIST);
    writeln('Team Batting Averages:');
    writeln;
    WRITELIST(x);
    x := TOTLEAGUEPLAYERS (READLIST);
    totplayers := GETINT(x);
    writeln('Total Active Players as of 15 June 82:');
    writeln;
    writeln(totplayers);
    if (totplayers > limit) then begin
      writeln('League exceeding authorized strength.');
      writeln('Notify Commissioner Immediately...');
    end;
#   include "errorlabel.i"
  end.  { program baseballstats }

**************************************************************
```

LIST OF REFERENCES

1. Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," Comm. ACM, v. 21 no. 8, p. 613-41, August 1978.

2. Henderson, P., Functional Programming Application and Implementation, p. 1-58, 126-127, Prentice-Hall International, Inc., 1980.

3. Curry, H.B. and Feys, R., Combinatory Logic, v. 1, North Holland Publishing Co., 1958.

4. Naval Postgraduate School Report NPS52-81-006, Values and Objects in Programming Languages, by Bruce J. MacLennan, p. 15-21, April 1981.

5. Chiarini, A., "On FP Languages Combining Forms," SIGPLAN Notices, v. 15 no. 9, p. 25-27, September 1980.

6. Grogono, P., Programming in PASCAL, p. 96-106, 223-239, Addison-Wesley Publishing Company, Inc., 1980.

7. Jensen, K. and Wirth, N., PASCAL User Manual and Report, 2d ed., p. 62-83, Springer-Verlag, 1978.

8. Jensen, R.W. and Tonies, C.C., Software Engineering, p. 337-356, Prentice-Hall, 1979.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
4.	Assistant Professor Bruce J. MacLennan 52Ml Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5.	Assistant Professor Douglas R. Smith 52Sc Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
6.	CPT O. D. Borcheller, USA 628 S. Buchanan Street Arlington, Virginia 22204	3
7.	CPT R. S. Ross, USA 533 Pine Avenue Pacific Grove, California 93950	3