A DATA DEFINITION FACILITY
FOR PROGRAMMING LANGUAGES

by

T. A. Standish

Carnegie Institute of Technology
Pittsburgh, Pennsylvania
May 18, 1967

Submitted to the Carnegie Institute of Technology
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This work was supported by the Advanced Research Projects
Agency of the Office of the Secretary of Defense (SD-146).
Abstract

This dissertation presents a descriptive notation for data structures which is embedded in a programming language in such a way that the resulting language behaves as a synthetic tool for describing data and processes in a number of application areas. A series of examples including formulae, lists, flow charts, Algol text, files, matrices, organic molecules and complex variables is presented to explore the use of this tool. In addition, a small formal treatment is given dealing with the equivalence of evaluators and their data structures.
**Table of Contents**

Title Page. ........................................... i
Abstract. ............................................. ii
Table of Contents. .................................... iii
Acknowledgments. ...................................... v
Chapter I. Introduction. .............................. 1
Chapter II. A Selective Review of the Work of Others. ........ 17
Chapter III. The Data Definition Facility. ................. 25
  1. Chapter Summary. .................................. 25
  2. General Description. .............................. 25
  3. Component Descriptions. ......................... 30
  4. Elementary Descriptors. ......................... 34
  5. Modified Descriptors. ............................ 40
  6. Descriptor Formulae. ............................. 48
  7. Declaring Descriptor Variables.
      and Descriptor Procedures. ...................... 51
  8. Predicates, Selectors, Constructors
      and Declarations. ............................... 53
  9. Constructors. .................................... 53
 10. Selectors. ....................................... 58
 11. Predicates. ..................................... 60
 12. Declarations. ................................... 64
 13. Reference Variables, Pointer Expressions
      and the Contents Operation. .................... 65
 14. Overlay Assignments, Sharing of Structures
      and Copying of Structures. ........................ 68
Table of Contents, Continued

15. Paths, Path Variables and Path Concatenation ........ 70
16. Other Operations on Descriptors ....................... 71
17. Parallel Assignments and Block Expressions ............ 72

Chapter IV. Examples and Applications ..................... 73
1. Formula Manipulation .................................... 73
2. List Processing ........................................... 95
3. Mapping Algol Text into Flow Charts ..................... 104
4. Electronic Circuits ....................................... 121
5. Complex Variables ....................................... 128
6. Files ....................................................... 129
7. Matrices ................................................... 134
8. Recognizers for Graphical Objects ....................... 137
9. Organic Chemistry Molecules ............................... 182

Chapter V. A Small Formal Study of the Equivalence Of Evaluators and Their Data Structures ........ 195

Chapter VI. Some Reduction Algorithms and Background Machines ........................................ 239

Chapter VII. Future Directions and Conclusions .............. 254

Appendix I. Additions to the Revised Algol Report Defining the Syntax of the Data Definition Facility ........ 267

Appendix II. Summary of the Semantics of the Data Definition Facility ........................................ 276

Appendix III. Data Structure Definitions ..................... 280

References ..................................................., 287
Acknowledgements

These early days of the science of computing, even if somewhat muddled, are filled with excitement and at Carnegie Tech especially the environment has been stimulating. This is due mainly, I feel, to the presence and leadership of a gifted, dynamic and personable triumvirate who have marshalled people, resources and, most of all, inspiring ideas in the construction of the computing enterprise at Carnegie. It has been both a unique privilege and an enormous pleasure to have had the opportunity to participate in this enterprise for the past four years.

I am particularly grateful to several individuals who have been helpful in the preparation of this dissertation.

First and foremost is Professor Alan J. Perlis. His outstanding lectures and teachings have provided many of the ideas that I find are now keystones in my thinking, and he has given generously of his valuable time and advice in helping me to prepare this dissertation.

My colleagues in the Formula Algol group: Renato Iturriaga, Rudy Krutar and Jay Earley, with whom I have shared the rewards and frustrations of sustained, constructive toil, are also deserving of mention for hours of interesting conversation and for their creative suggestions.

I am also grateful to Professor Allen Newell whose remarkable originality has been a broadening influence and
a stimulus in getting me to look at different approaches to the work of this dissertation.

As a recipient of a National Science Foundation fellowship during my graduate studies it is a pleasure gratefully to acknowledge my indebtedness to the National Science Foundation for financial support. Finally, computer time and facilities were provided in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contract SD-146 to the Carnegie Institute of Technology.
Chapter I

Introduction

This dissertation presents a method for defining and manipulating several varieties of data structures. This method consists of embedding a descriptive notation for data structures within a programming language in such a way that the resulting language behaves as a synthetic tool for describing and constructing data and programs in a variety of application areas. A series of examples are presented which explore the use of this tool and a small formal presentation is given dealing with the equivalence of evaluators and their data structures.

One hope for the dissertation is that the data structure notation it presents will serve both a descriptive and an integrative role. Enough examples have accumulated in our programming experience of different kinds of useful and durable data structures that it becomes worthwhile to study them deliberately and to search for a notation which, to some extent, summarizes the variations in structure that they represent. What is desired is a notation whose permissible variations allow us to describe a wide range of different data structures including, as subsets, the useful structures known to us and new ones as well. A good descriptive notation should not only
be a summary of what is known, it should lead along natural lines of generalization suggested by its combining laws and rules of growth to the description of new structures. With regard to its use as part of a programming technique for describing and manipulating data, the notation should permit descriptions of data structures to be concise, clear and direct.

In our view, the significant problems that we face in dealing initially with new areas of research in computer science are "problems of discovery, formulation, representation and immediate generalization [45]" and that with regard to practical application "we are not [during the initial stages of exploration] at the place of building very elaborate or formal mathematical structures that are significant [45]". The study of data structures has perhaps advanced beyond this initial stage. At present, we have many results of the processes of discovery and formulation (e.g. lists, arrays, strings, rings, formulae, trees and directed graphs). These discoveries make possible a process of immediate generalization. The data structure notation presented in this dissertation exhibits one immediate generalization of known data structures. Thus the notation summarizes, integrates and provides descriptive power. It also points in new directions.
Giving a notation of broad descriptive power enhances our opportunities for creating meaningful formal mathematical structures. A small formal treatment of the equivalence of evaluators and their data structures, given in Chapter V, although not of broad practical significance, is intended as a harbinger of what becomes available to us in the realm of mathematical formalization. Ultimately, of course, we hope that significant mathematical structures can be formulated which will summarize the significant discoveries about data structures with precision and generality. At this point the study of data structures will approach maturity.

Another objective of the dissertation is to provide a method for improving programming languages. One of the constant concerns of computer science is to design good programming languages, and, once they are implemented, to enrich them and improve them, for programming languages are a principal means whereby we specify and control the behavior of computers and whereby we can organize our transactions with computers to achieve multiplication of effect without multiplication of effort.

For a given programming task, the greater the ease with which a programmer can describe in a given language the data of
interest and the greater the ease in that language with which he can manipulate the data he has described, the greater the usefulness of that language for that task. Conversely, a programming language is relatively less useful for a given task to the degree that it becomes awkward, indirect and complex to formulate and manipulate representations of the data of interest to the task in terms of the primitive data and primitive operations supplied in that language. Indirectness of data representation and data manipulation may cause a loss both of clarity of structure and also of efficiency since operations on the representations which could be expressed or performed once prior to writing an algorithm may be distributed through it and since the primitive data structures used to represent the data of interest may not be well adapted to the task. One sees here that our concern is not so much with whether it is possible to perform a task within a programming language but rather whether it can be done with convenience, clarity and efficiency. As Perlis puts it in his Turing Lecture, "Programmers should never be satisfied with languages which permit them to program everything, but to program nothing of interest easily.[48]" These considerations are economic in nature.

This dissertation hopes to demonstrate, by means of a series of examples, that embedding a data definition facility in a language
permits it to describe and manipulate data with conciseness, clarity and grace over a diverse range of application areas. Success, we feel, is to some degree to be measured by the clarity of structure achieved in describing and applying appropriate programming techniques to a diversity of problems.

In order better to see why adding a data structure notation to a programming language is likely to improve it we examine briefly a small selection of present day programming languages with regard to the primitive data and operations they offer. In doing so we intend to expose a weakness of these languages with regard to their versatility in describing data structures. Algol [43], to begin with, contains real numbers, integers, logical values, strings, and arrays of arbitrary dimension, as primitive data structures. Its operations include the arithmetic, logical and string operations as well as accessing and assignment functions for arrays. Lisp [40] contains atoms and composite symbolic expressions, called s-expressions, as its primitive data structures, and it contains a composition operation cons[x;y], which operates on atoms and s-expressions to produce s-expressions recursively. Formula Algol [49] adds to the primitive data structures of Algol both formulae and list structures, and adds a set of operations to manipulate them. Cobol [14] operates on files as one of its primitive
data structures with a set of file manipulation operations. Another example is an order code for a digital computer whose primitive data structures might be bit patterns interpretable as numbers, logical constants, addresses, instructions and so forth.

Each of these languages is natural to use for a certain area of applications. For example, one might choose Formula Algol if one wished to differentiate accurately a long, complicated formula. Formal differentiation could, of course, be done in Lisp[60], in Algol [64], or even in Cobol. However, to accomplish this, the formula data structures which must be transformed and derived in the process of differentiation would have to be represented synthetically in terms of appropriate combinations of the primitive data structures of Lisp, Algol or Cobol respectively, and the composition, accessing and testing functions over these representations of formulae would in turn have to be represented as combinations of the primitive operations on primitive data structures in these three languages. An examination of the problems of representing formula manipulation in Algol, Lisp or Cobol reveals that these media are, to a limited extent, strained in their representational powers. The task is certainly not impossible; it is merely inconvenient and unnatural and requires indirectness in the use of primitives of the language to accomplish operations that may be
evoked directly in Formula Algol. Why is this so? First of all, the amount of space consumed during the execution of a particular formula manipulation algorithm depends on the size and number of constructions of representations of formulae evoked during that execution, and it is difficult to predict in advance exactly how much space will be required for a given construction. Thus, in order to represent formulae naturally, a language should possess at least one variable data structure whose capacity can be determined at execution time and which need not be fixed in advance. In the case of Algol, there are two such structures. Own arrays provide one if we exit and reenter the blocks in which they are declared, and under the assumption that in Algol integers can be of unbounded size, Gödel numbering provides another. Both of these structures are unnatural to use in Algol as are representations of formulae dependent on a number of fixed arrays. If one could declare arrays in Algol with indefinite upper bounds, the situation would be improved. Lisp has a variable data structure in the form of lists but lists must be used for everything. In particular, arithmetic is made indirect and one is forced in doing arithmetic to go through a double layer of evaluation: the Lisp evaluator followed by an arithmetic interpreter. In other words, the single data structure of Lisp is not well adapted to all applications. [The appearance of Lisp 2 [1] seems to constitute a recognition of this.]
In the work of Fenichel [21], Engleman [17] and Moses [41] Lisp is used as a background machine, and both syntax analyzers and data mapping functions are interposed between Lisp and the user to translate from representations that are well adapted to the task of formula manipulation into the data structures of the background machine. Lisp is sometimes considered by programmers as a foreground machine, but, in fact, is frequently used indirectly (and perhaps inefficiently) as a background machine.

This discussion is not intended as a glorification of Formula Algol. It would, of course, be strained and unnatural to maintain and update information in an employee file in Formula Algol. Here Cobol supplies primitive data and operations better adapted to the task. The point to be made is that to each language there may be associated classes of problems with which it is within its purview to solve with grace and convenience, and that for each there are classes of problems which lie beyond, requiring for their solution indirect and inefficient use of the basic powers of the language.

One difficulty we face increasingly in present day programming, is that problems arise involving widely varying classes of data requiring for their solution the combined advantages of several kinds of processing to be available in one language. D. Bobrow's "Student"
program [7], which solves algebra word problems, is a good example of this. Bobrow's program uses a simulated string manipulation processor (Meteor[5]) to accept and parse algebra word problems stated in a subset of English. The result of sentence analysis is the formulation of a set of simultaneous equations requiring for their solution formula and arithmetic processing. Lisp is used as a substrate for the entire system.

One solution to the requirement that a language be general and that it combine the advantages of various kinds of processing in one language is to construct a language which is to some extent the union of several others in the framework of a common set of conventions and control structure. This is the approach of PL/I [50], and to a limited extent of Lisp [2] and Formula Algol [49], three languages which attempt to be eclectic. In the first place, the kind of generality offered by these languages is costly in space and time. Such systems tend to be large, which is not so bad if the computer on which they are run has a large, fast backup storage and if the presence of only one part of the system is required at any time, but perhaps worse, they tend also to be slow since the enormous variety of constructions permitted in the source language implies more selection is necessary to decode a given program and this costs time during translation.
Our experience with Formula Algol shows that few programs, if any, utilize the full generality of the language, yet this generality is available everytime a program is run. Consequently, the typical user, in the absence of a mechanism to specify a subset of the language, pays space and time for generality he doesn't need. Of course, the wise thing to do is to provide the user with a mechanism for specifying a subset of the language of interest to him. Then he uses and pays for only as much generality as he needs.

A problem still exists, however. The eclectic language which is a union of other languages has only partial generality. Applications arise where the fixed set of primitive data and operations supplied initially are not well adapted. (E.g. Formula manipulation in PL/I or Lisp 2, and file manipulation in Formula Algol). Perhaps the lesson to be learned is that in designing a language one cannot foresee all applications to which the language will be put, and consequently it is not possible to build into a programming language in advance the appropriate set of primitive data structures and operations that will serve all possible applications well, for in the act of choosing a specific set of primitives and leaving them fixed for all applications, the freedom of choice to define primitives well adapted to the task at hand is lost. Of course, in any programming language we must always start with primitives, but making the set of primitives initially large does not
make the set adaptable and flexible. Here, as Perlis notes in his Turing Lecture, we should profit from our experience with functions.

Our experience with the definition of functions should have told us what to do: not to concentrate on a complete set of defined functions at the level of general use, but to provide within the language the structures and control from which efficient definition and use of functions within programs would follow. [48]

Thus we contend that one of the things we must do in order to achieve better flexibility in programming languages is to provide a definition facility that allows a programmer to define data structures that are well adapted to a given intended application. This definition facility should permit the programmer to describe with ease the data structures that are critical to a task and to formulate with ease the operations over these data structures that are necessary for describing the processes he intends to carry out. Therefore, it makes sense to embed a descriptive notation for data structures within the context of a programming language creating a data definition facility whose use as a synthetic tool may be explored and whose contribution to the improvement of the language in which it is embedded may be measured. We hope to show that such a facility when added to a programming language can significantly improve its versatility with regard to the number of application areas it can handle with convenience and can do so in such
a manner that the user need not pay space and time for generality he doesn't need.

This definition facility will attempt to organize the description of data in such a manner that the definitions of structures that are well adapted to an intended application may be made once before writing the algorithm for that application, permitting the use of operations and terminology in the algorithm that are related directly to the structure of the defined data, rather than always operating indirectly upon representations of the data of interest by means of operations and terminology related to the fixed set of initial primitives from which the representations have been synthesized. What is desired is that indirectness in dealing with the data of interest be reduced for the programmer and that the burden of administrating representations be passed on to the machine.

This dissertation, of course, has its limitations. One of the important problems we face in programming is that of finding representations which satisfy the requirements of a given programming task. Given a set of behaviors or properties that a representation must have in order to satisfy the requirements of a task we know of no way to generate new representations to satisfy them. We do not even know of any very effective way to state accurately and formally the constraints that we
would like representations to obey. A mature science of computing must contain a theory of representation which addresses itself to these questions. Concomitantly, a mature theory of representation must discuss questions of efficiency and optimality of representations. While we recognize these as important problems we must confess that they are beyond the scope of this dissertation.

Having stated our objectives and our point of view we turn to a more precise formulation of what we mean by providing a data definition facility for programming languages. It would be theoretically pleasant to formulate the data definition facility in a manner somewhat independent of particular programming languages. We might consider that each programming language defines a machine for which programs may be written. For example, an Algol translator is, in effect, a device that transforms the raw computing power of a computer into a machine which accepts and executes Algol programs. Furthermore, it is theoretically true that any programming language which is powerful enough to simulate a Turing machine can be used, more or less efficiently, to perform any program that can be performed in another Turing machine equivalent language. Hence, disregarding, for the moment, efficiency and convenience, and concentrating only on the question of whether or not tasks are possible, programming languages which are Turing machine equivalent have equal representational power.
All such languages, then, qualify as theoretically possible background machines with which to represent the data of interest to a given task. The task of formulating an adequate data structure definition facility can be specified as follows. We are given three things: (1) a background machine, (2) a set of data, and (3) a set of processes we intend to carry out on the data. The background machine may be Algol, it may be Lisp, or it may be a Turing machine. Theoretically we don't care what it is; practically, of course, we do. As examples of data spaces and processes we intend to carry out, we might be given simple Algol text and flow charts as data and we might be asked to map the Algol text into the corresponding flow chart, if there is one†, or vice versa. Or we might be given organic chemistry molecules as data and we might be asked to determine stereoisomers, if any exist, or to find Geneva names. Thus, given a background machine, a set of data and a set of intended processes to carry out on the data, an adequate data definition facility is specified provided that by use of that facility we can (1) describe the data with ease, and (2) obtain with ease a set of primitive operations appropriate to the data space described from which to compose the processes to be carried out, and provided that (3) the facility can supply automatically from these descriptions of the data space in terms of the primitive data structures and operations of the background machine.

---

† Christopher Strachey has exhibited a procedure in CPL that cannot be translated into a flow chart (cf. [62], section 9.3, p. 2).
composites of its primitive data structures representing objects
in the described data space, and composites of its primitive
operations representing primitive operations on the data space.

This dissertation, among other things, attempts to exhibit
a data structure definition facility and associated reduction algorithms
that will solve the above problem for the following data spaces and
for the following background machines.

Data Spaces  Background Machines
1. Formulae  1. An Abstract Machine
2. Organic Chemistry Molecules  2. A List Processor (Lisp)
3. Electronic Circuits  3. A Formula Manipulator
4. Algol Text  (Formula Algol)
5. Files  4. A Machine Language
6. List Structures  (CDC G-21)
7. Flow Charts
8. Matrices
9. Complex Variables
10. Recognizers for Objects
    in Two Dimensions

Since descriptions of data spaces are reduced into data
representations in terms of primitives of a given background machine
and since from the descriptions are derived automatically both the
primitive operations over the data space and representations of these
primitive operations in terms of an equivalent composition of primitives
in the background machine, the definition facility and its associated
reduction algorithms described in this dissertation constitute a modest first step toward the goal of automating the representation of data structures by programming languages.
Chapter II

A Selective Review of the Work of Others

A number of significant ideas have appeared in the literature concerning mechanisms of use in defining data structures. These ideas are both useful and good, and the data definition facility introduced in Chapter III employs an abstraction of them. This chapter reviews a selected portion of them and develops this abstraction.

The first idea is the idea of the plex, which originated with Ross [54]. Essentially a plex is an n-component element. "The components of an element may be of any form. They may be one-bit quantities, machine addresses, machine instructions, symbolic information, or numerical data in any appropriate number form. In particular, (and this is the means whereby the system includes all other known symbol manipulation schemes), a component of an element may be a 'link' or reference to another element" ([54], p. 147).

Plexes are similar to the basic data structures of two more recent languages, Coral [52] and L6 [31].

McCarty is responsible for two important developments. First he described formally the data space of Lisp as being composed
of atoms and s-expressions. After defining atomic symbols ([39], p. 186) he states ([39], p. 187): "S-expressions are then defined as follows: 1. Atomic symbols are S-expressions, 2. If \( e_1 \) and \( e_2 \) are S-expressions, so is \( (e_1 \cdot e_2) \)." This was one of the first formal definitions of the data space of a programming language, and we feel that the elegance of Lisp is, in part, due to the clarity of this definition. We hope to show in later examples that similar elegance is available in the definition of other kinds of processing, such as formula manipulation, as a consequence of similar clean definitions of data structures, and that this property is not unique to Lisp.

Second, McCarthy defined a mechanism to be added to Algol to permit the formal definition of new data types. Given that \( S_1, S_2, \ldots, S_m \) are well defined subspaces of the data space of the language, one can form Cartesian products from them. For example: \( S_{1_1} \times S_{1_2} \times \ldots \times S_{1_n} \). Elements of the resulting product space are ordered \( n \)-tuples of elements chosen from the \( S_{1_k} \), \( 1 \leq k \leq n \). Thus new elements can be formed by composing \( n \)-tuples of components chosen from designated subspaces of the data space of the language, and these components may be selected from a given element by using the projection functions that map an \( n \)-tuple onto its \( i \)th component. In McCarthy's scheme a method is given for naming the projection functions, for naming the constructive function that forms an \( n \)-tuple from \( n \) designated components, and for naming the type of the data space of the result. For example, he introduces the complex numbers with the following
declaration ([38], p. 45):

    cartesian complex, ((realpart, real), (imagpart, real)), mkcomplex;

Complex multiplication can then be defined with the following procedure:

    complex procedure cmult(u, v); complex u, v ;
    cmult := mkcomplex( realpart(u) \times realpart(v) - 
                      imagpart(u) \times imagpart(v), 
                      realpart(u) \times imagpart(v) + 
                      imagpart(u) \times realpart(v) ) ;

McCarty also introduces a declaration called union which defines
a new data type, say t, in terms of alternate data types defined by
Cartesian products or chosen from other sub-data spaces. Given
a datum of type t one can apply a function to it to determine the type
of the subspace from which it came. The projection functions of
that subspace may then be applied to the datum to access its components.

Another important mechanism is the description list of IPL
as formulated by Newell, Shaw and Simon [44], or the equivalent
property list of Lisp[40]. A description list X is a sequence of pairs.
The first member of each pair is called an attribute, and the second member
of each pair is called a value. Thus the generic form of X is :
X = (atr_1, val_1)(atr_2, val_2)...(atr_n, val_n) . Given the name of a description
list L and the name of an attribute on it, say A, one may retrieve the
value V associated with the attribute A on L. This process of accessing
the value V by means of a search on L with the name A is similar
to the process of accessing the i-th component of a Cartesian n-tuple
by use of the name of the projection function. However, the description
list is more variable in the sense that more attribute-value pairs
may be appended to it whereas once storage for an n-tuple has
been allocated the length of the n-tuple is fixed unless storage is
reallocated. Accessing the description list is also less efficient.
Thus one pays for an increase in the capacity to sustain variation
with a decrease in efficiency. We will hereafter use the word
variability to refer to the capacity to sustain variation. (This term
was introduced by Perlis in his Turing Lecture[48] and in his
NATO Lecture Series in Edinburgh, 1966 [unpublished]). Since
description lists may contain other description lists as values of
attributes, composed data structures may be constructed from them
of considerable flexibility. In fact, GPS 2-5 was programmed entirely
with description lists as its primitive data structure †.

In what we consider to be a most important contribution,
Landin introduced the notion that one could associate with the description
of a data structure, which he termed a "structure definition"[34],
three classes of operations: predicates, selectors and constructors.

† Remark in a lecture by A. Newell
A structure definition specifies a class of composite information structures, or constructed objects (COs) as they will be called in the future. It does this by indicating how many components each member of the class has and what sort of object is appropriate in each position; or, if there are several alternative formats, it gives this information in the case of each alternative. A structure definition also specifies the identifiers that will be used to designate various operations on members of the class, namely some or all of the following:

(a) **predicates** for testing which of the various alternative formats (if there are alternatives) is possessed by a given CO;

(b) **selectors** for selecting various components of a given CO once its format is known;

(c) **constructors** for constructing a CO of given format from given components.

An example of a structure definition for the data space of lists taken from Landin ([34], p. 312) is the following:

A list is either null or else has a head \( h \) and a tail \( t \) which is a list.

Landin uses `constructnullist` and `constructlist` to designate the constructors of the two alternative formats of lists. The first function takes no arguments and the second takes two arguments. The functions \( h(x) \) and \( t(x) \) are selectors which take the head and tail of the list \( x \) respectively. The functions `list(x)` and `null(x)` may be used as predicates. One now sees that McCarthy has arrived
implicitly at similar correlations between data space definitions and the associated set of predicates, selectors and constructors. McCarthy's projection functions are, in fact, selectors. He names his constructors explicitly in his Cartesian declaration, and his type tests are predicates. Thus one sees in the stylized English notation of Landin the same ideas repeating themselves in different dress as are found in the Cartesian product proposal of McCarthy. Theplexes of Ross, the description lists of Newell, Shaw and Simon, the BNF grammars of Backus [3], and the heterogeneous vectors of Galler and Perlis [24] all display the similarity that the alternative composite structures in each are formed from an aggregate of components and it is natural to define for these composite structures the selectors that access the components and the constructors that compose them. Where there is more than one format defined it is natural to define predicates to distinguish between them.

Let us now list the critical similarities and important ideas that can be distilled from these examples. (1) Data structure definitions may provide a name for the data type being defined. (2) This defined data type names a data space which may be composed of alternative types of composite structures (if there are alternatives). (3) These composite structures may bear names (but not necessarily) and their internal structure is specified by what amounts to specifying the name or ordinal position of each part and a corresponding specification
of the data space from which it is appropriate to choose the given part. (4) A data type word (in one to one correspondence with a data space) may be used to specify the data space from which a part is chosen. (5) Recursion is permitted so that a part may be specified as belonging to the data space being defined.

Except for the trade off between efficiency and variability it matters not whether a composite structure is specified by means of (1) Cartesian n-tuples with named projection functions used to specify the parts, (2) description lists with attribute names used to specify the parts, (3) n-word plexes with field names used to specify the parts, (4) nodes in a graph with edge names leading away from the node used to specify the parts, or (5) heterogeneous vectors with integer subscripts used to select the parts [24]. For all of these are behaviorally isomorphic with respect to the act of composition and the act of selection or accessing. That is, given a notation for constructor functions and selector functions, we can specify interpretations of this notation by defining functions in each of the above systems in such a way that no matter which underlying system we have chosen the notation evokes operations which behave the same way. The choice of which of the above systems to use could be said to be influenced by the choice of
a background machine. For an assembly language background machine one might choose plexes and field name selectors; for an Algol background machine one might choose vectors and subscript selectors; for a list processing background machine one might choose description lists with attribute selectors. The point is that because of the behavioral isomorphism with respect to composition and accessing certain types of data structure descriptions may be formulated independently of background machines and their associated primitives. By means of reduction algorithms these data structure descriptions may be translated into representations of data spaces and their associated operations in terms of the primitive data and primitive operations of a given background machine. In this dissertation we first formulate data structure descriptions independently of background machines and then show how they can be reduced to representations in the primitives of four given background machines.

This chapter would be incomplete were we not to mention several additional interesting references devoted to or germane to the study of data structures. Among the better of these, from which it is felt that some profit might be extracted, are Hoare[26], Holt [27], D'Imperio [15] and Lindsay and Pratt [37].
Chapter III
The Data Definition Facility

1. Chapter Summary

This chapter presents the data structure definition facility as an extension to Algol 60 [43]. The facility is explained by enumerating its various constructions, by listing their properties and by giving examples. Algol 60 was selected as a parent language in which to embed the facility because of the grace with which Algol can accept extensions and flourish under surgery, and because the cleanliness of its description permits corresponding cleanliness in the description of extensions. The essential features of the data definition facility presented here could have been added to other languages as well for the facility does not depend in substance on features of the Algol model. Three appendices related to this chapter are: Appendix I, which gives the syntax of the data definition facility as an addition to the Revised Algol Report [43]; Appendix II, which gives a summary of the semantics of the facility; and Appendix III, which lists for comparison and reference the various data structure definitions employed in the dissertation.

2. General Description

The basic ideas of the data definition facility are as follows. A new class of expressions, called descriptor expressions, is added
to Algol 60. The value of a descriptor expression is a descriptor formula (or descriptor for short). A descriptor describes a data space, i.e. it describes the static structure of a class of data forming a domain of variation over which the values of declared variables may range. A descriptor may be specified not only by evaluating a descriptor expression but also by declaring and calling a descriptor procedure which may take parameters and compute as value a descriptor whose form and composition depends upon these parameters.

Descriptions of data spaces may thus be programmed.

A descriptor contains enough descriptive information to determine predicates, selectors, constructors and storage allocation policies for data in the data space it describes. Given a background machine having at least one type of composite structure whose capacity need not be specified in advance of allocation time and whose components may include references to other data, we specify a set of algorithms, which we have chosen to call reduction algorithms, which associate with each descriptor a data representation in the background machine and which provide for this data representation background machine representations of its predicates, selectors and constructors.

The act of constructing a representation of a data structure involves, among other things, allocating space in the background machine
so as to form a representation of the format specified by the descriptor for the data structure. In order to be faithful, this act of allocation must select and compose background data structures which model the capacity and connectivity specified by the format of the descriptor. (Examples of how this can be done are specified in Chapter VI.) The selectors for an allocated structure are then represented by appropriate compositions of background machine accessing functions valid for the allocated structure. Likewise, predicates on allocated composite structures are represented by appropriately programmed compositions of predicates for the parts, which reduce ultimately to calls on primitive atomic predicates supplied in the background machine.

In the data definition facility, notations are provided as extensions of Algol 60 to call these predicates, selectors and constructors for the purpose of describing processes over the data space defined by a descriptor, and descriptor names may be used as declarators to declare variables which take on values from the data space associated with the named descriptor. Thus the data definition facility is so organized that the creation of a new data space and its associated predicates, selectors, constructors and
declarators involves no more than writing or computing its appropriate descriptors.

Descriptors are of two basic sorts: elementary descriptors and modified descriptors. Elementary descriptors provide descriptions of the alternative data formats (if there are alternatives) as compositions of component descriptions. The composition operations include iterative, recursive and nested composition. A single component description describes the data space from which the described component may be chosen, and, in the case that the component description bears a name, the name may be used as the name of the selector function that selects the described component from a datum constructed according to the described format. If an elementary descriptor bears a name, this name may be used as the name of a predicate which tests whether a given datum has the format described by the descriptor. Further, an elementary descriptor may be used as an argument to a functional which maps it into a constructor function, which constructor function may be used to allocate space and to construct data of the format described by the descriptor.
A modified descriptor has a modificand, which is an elementary descriptor or descriptor valued expression, and one or more modifiers. These modifiers are used for one or more of the following purposes: (1) to alter, to expand or to restrict the format described by the modificand, (2) to describe an extension to the predicate associated with the modificand, or (3) to describe an extension to the constructor associated with the modificand. A descriptor formula is, in general, either an elementary descriptor or a modified descriptor or a disjunction of elementary and modified descriptors.

In addition, the following classes of operations exist within the data definition facility. References to substructures of allocated constructions may be computed and assigned as the value of reference variables. References may also be used in the construction of data structures and descriptions of how these references are to be assigned may be contained in constructor modifiers permitting the description of richly connected data by descriptor formulae. Access paths may be derived from operations on descriptors and representations of these paths may be stored as values of path variables. Paths may be combined and may be applied
to data with selector operators. Functions exist to permit the sharing of a substructure by two or more superstructures, to permit the copying of structures, and to permit shared substructures to be replaced or overlayed by others so that all references to the shared structure are made to point to the replacement. Policies of initialization may be defined to specify the initial values of declared variables. Finally, an abbreviation device is introduced to contract the definition of character sets.

3. Component Descriptions

A component description specifies the data space from which a component of a datum may be chosen, and if the component description is named, its name may be used to specify the selector function that selects the described component from a datum constructed according to the format given by the elementary descriptor in which the given component description occurs. For example, the descriptor formula for a data space of complex variables might be given as follows:†

\[
\text{elementary descriptor} \quad \text{complex} \leftarrow [ \text{realpart} : \text{real} | \text{imagpart} : \text{real} ]
\]

\[
\text{descriptor variable which is assigned a descriptor formula as value}
\]

\[
\text{figure 1.}
\]

† In this example, and in the sequel,'→' has been used as an assignment operator in place of the Algol character pair ':='.
Here, the descriptor formula for a complex variable consists of a single elementary descriptor which is composed of two component descriptions. The first component description is 'realpart:real' and the second component description is 'imagpart:real'. The names of these components are respectively 'realpart' and 'imagpart'.

The word 'real' is used as the data space specifier of each component signifying that the data space from which each component is to be chosen is the data space of representations of real's given in the particular background machine in which the data structures are to be represented. If u is a complex variable, then realpart(u) selects the first component of its value, which is of type real, and imagpart(u) selects the second component of its value, which is also of type real.

Thus, component names serve the dual purpose of naming component selector functions. Another example of a descriptor is

\[
cpair ← [ \text{complex} | \text{complex} ]
\]

Here, both component descriptions of a cpair are given by the value of the variable 'complex'. These component descriptions do not bear names. A third example is:

\[
\text{descriptor procedure atom(n) ; value n ; integer n ;}
atom ← [\text{name:identifier} | \text{valence: (=n)} | \text{bonds: Sequence(n, any)}]
\]
In this example the descriptor formula for an atom is computed by a procedure with a parameter which is the valence number of the atom. The descriptor formula computed will consist of an elementary descriptor with three named component descriptions for: name, valence and bonds. The data space from which the name must be chosen is the data space of representations of identifier's in the chosen background machine. The data space from which it is appropriate to choose the valence is a unit data space, consisting of only one element, described by the expression \( n \).

An expression \( E \) preceded by a unary equal sign may be used to specify the unit data space which is the value of that expression. The data space specifier of the bonds component is the expression 'Sequence(\( n, any \))'. This is a procedure call which computes a descriptor to describe the data space of the bonds component. The descriptor computed depends on the current value of the parameter \( n \).

(For the procedure declaration for Sequence(\( n, any \)) is given in Section 4.)

A fourth example is

\[
\text{sumformula} \leftarrow \{ \text{operator:}(\text{'+'})\text{operand: Sequence}(k, \text{formula}) \}
\]

This example shows that a data space specifier may be a quoted
character string, e.g. '++'. Quoted character strings of any length are permitted. A final example is

\[ \text{Positive Integer} \leftarrow \{ \lambda (x), x > 0 \wedge \text{integer}(x) \} \]

Here, the descriptor for a PositiveInteger is an elementary descriptor consisting of one component description. The data space specifier of this component description is a monadic \( \lambda \)-expression [12, 34] defining the set of representations of positive integers in the chosen background machine. A data space specifier may also be the symbol 'nil' which signifies the empty data space and which serves as an identity under computed compositions of descriptors, as will be explained in Section 4.

In summary, a data space specifier is either

1. A type word (e.g. real, identifier, any, reference),
2. An expression preceded by a unary '=' (e.g. =n),
3. A quoted character string (e.g. '+', 'KdT-'),
4. A descriptor valued expression (e.g. complex, Sequence(n, any)),
5. A monadic, Boolean \( \lambda \)-expression (e.g. \{ \lambda (x), x > 0 \wedge \text{divisible}(x, 2) \}, or
6. The symbol 'nil' (e.g. nil).

\[ - 33 - \]

† A monadic \( \lambda \)-expression is one having one and only one bound variable.
A **component description** is either a **data space specifier** (e.g. `real`), or is a name followed by a colon followed by a data space specifier (e.g. `realpart : real`).

### 4. Elementary Descriptors

In essence, elementary descriptors provide pictorial format descriptions of data structures by composing component descriptions under one of several forms of composition among which are iterative composition, recursive composition and nested (or tree structured) composition.

**Nested Composition**

One way of specifying an elementary descriptor is to write directly within square brackets the sequence of component descriptions that compose it, separated by vertical strokes (e.g. `[ realpart: real | imagpart : real ]`). Since component descriptions are permitted to be descriptor valued expressions, nested (or tree structured) descriptors may be specified (e.g. `[[real|real]|cpair]`).

**Iterative Composition**

Replication of structure is one of the basic forms of organization in data structures. The descriptor `n × [C]` describes
a data structure which consists of n replications of the component described by C. Thus it is equivalent to the descriptor \[ C | C | \ldots | C \]. A more potent form of replication permits parametric variation in the replicated component descriptions. This type of replication is denoted by expressions of the form \( i \rightarrow n \times [C(i)] \) where i is a controlled variable that takes on the values from 1 in steps of 1 until n, and where the \( i^{th} \) component description \( C(i) \) is computed as a function of this controlled variable. Thus \( i \rightarrow n \times [C(i)] \) is equivalent to the elementary descriptor \[ C(1) | C(2) | \ldots | C(n) \].

An example of the (nested) use of replicators in defining lower triangular arrays of m rows of type t is given by the following procedure.

```plaintext
descriptor procedure LowerTriangularArray(m, t);
    integer m ; descriptor t ;
    LowerTriangularArray \leftarrow i \rightarrow m \times [i \times [t]] ;
```

When it is desired to specify a class of replicated structures where the number of replications is not known in advance of the allocation or construction time for the data structure the construction `indefinite \times [C]` may be used. This specifies a data structure with
an indefinite number of replications of components of type C.
The constructor function corresponding to descriptors of this
form takes a variable number of parameters. When this constructor
function is applied to a list of n parameters of type C, storage
is allocated for a data structure of n components of type C. Thus
the number of replications of C becomes known and specified at
allocation or construction time rather than at the time the description
of the data space is provided.

The replication device presented in this section is an
extension of a similar device given by Perlis in his format language
for input and output in Algol [47].

Recursive Composition

Recursion is another basic form of organization in data
structures and recursive description permits one to describe
data structures of indefinite extent in finite closed form. Recursive
composition involves merely the use of the name of a descriptor within
the descriptor expression that defines it. An example is

\[
\text{BinaryFormula } \leftarrow \text{ atom } \lor \left[ \text{ Operator:ArithOp} \right. \\
\left. \text{ LeftOperand: BinaryFormula } \mid \text{ RightOperand: BinaryFormula} \right];
\]

The construction ' .BinaryFormula ' is an example of the use of a name
of a descriptor variable within an elementary descriptor. The name of
the descriptor variable 'BinaryFormula' is signified by prefixing it with a dot '.' as in Formula Algol [49]. If a variable is not dotted it signifies that the value of the variable is to be used rather than the name. The dotted variable '.BinaryFormula' is used within the right-hand disjunct of the descriptor formula defining a BinaryFormula to designate recursively that the LeftOperand and RightOperand components of a BinaryFormula may themselves be BinaryFormula's.

Computing Elementary Descriptors

An elementary descriptor may be computed and assigned as the value of a descriptor variable or descriptor valued procedure identifier. For example, the following procedure computes a descriptor:

```pascal
descriptor procedure Sequence(n, t); value n, t; integer n;
descriptor t;
begin descriptor temp; integer i;
temp ← nil;
for i ← 1 step 1 until n do temp ← [ t ∩ temp ];
Sequence ← temp;
end;
```

In the body of this procedure an elementary descriptor is computed
and assigned as the value of the procedure identifier 'Sequence'.

Concatenation of elementary descriptors is denoted \( y \odot z \).

For example, if \( y = [\text{real} \mid \text{real}] \) and \( z = [\text{integer} \mid \text{integer}] \)
then \( y \odot z = [\text{real} \mid \text{real}] \odot [\text{integer} \mid \text{integer}] = [\text{real} \mid \text{real} \mid \text{integer} \mid \text{integer}] \).

The symbol \( \text{nil} \) is an identity element both under concatenation
of elementary descriptors and under composition of component
descriptions with the vertical stroke operator '|'. Specifically,
if \( X \) is an elementary descriptor then \( X \odot \text{nil} = \text{nil} \odot X = X \), and
if \( Y \) is a component description then \( [Y \mid \text{nil}] = [\text{nil} \mid Y] = [Y] \).

Thus the value of the function designator call \( \text{Sequence}(3, \text{integer}) \) is
the elementary descriptor \( [\text{integer} \mid \text{integer} \mid \text{integer}] \).

Notice that the variable \( \text{temp} \) contains non null elementary descriptors as
values on the second and third cycles through the body of the \textbf{for}
statement in the Sequence procedure and that on each cycle through
the \textbf{for} statement a copy of the descriptor \( t \) is concatenated with
the previous value of \( \text{temp} \). If the statement \( \text{temp} \leftarrow [t \odot \text{temp}] \)
in the procedure above had been \( \text{temp} \leftarrow [t \mid \text{temp}] \), instead, then
\textit{nested} rather than \textit{linear} composition of component descriptions
would have resulted so that, for example, the function \( \text{Sequence}(3, \text{integer}) \)
would have had the value \( [\text{integer} \mid [\text{integer} \mid \text{integer}]] \). Thus
a choice of \textit{linear} versus \textit{nested} composition may be made
by using the stroke operator for nested composition and
the concatenation operator for linear composition.

A shorter way to write the procedure \( \text{Sequence}(n, t) \) is by using the replication operator as follows:

\[
\text{descriptor procedure } \text{Sequence}(n, t); \text{ integer } n; \text{ descriptor } t;
\]

\[
\text{Sequence} \leftarrow n \times [t];
\]

In fact, we see that the descriptor \( n \times [t] \) is a synonym for \( \text{Sequence}(n, t) \), as well as for the construction \( [t | t | \ldots | t] \).

Thus replicators take care of the case when identical components are repeated and the case when components whose description varies depending on the values of unit stepping controlled variables are repeated. Any descriptor with more variation than this must either be written out in advance or if its form is not known in advance then a computation must be specified which computes it as a function of quantities given at run time, during the execution of a program. Replicators are therefore merely an abbreviation for a special class of descriptors that can be computed or written by more fundamental and general means, but they are an important class of abbreviations for the following reason. If we compare the space required to store a replicator of the form \( n \times [m \times [C]] \) with its equivalent expansion \( [C | C | \ldots | C] | [C | C | \ldots | C] | \ldots | [C | C | \ldots | C] \)

\[
\text{n-times}
\]

\[
\text{m-times}
\]
we see that the replicator formula provides a considerable saving in space especially as the size and number of dimensions of replication is increased. In the case of an m by n rectangular array the expanded descriptor formula is essentially a 'picture' of the format of the array, and the replicator is a contraction of this format picture.

5. Modified Descriptors

A modified descriptor has a modificand, given by an expression whose value is an elementary descriptor, and one or more modifiers given as follows:

a. The Format Modifier

A format modifier alters, extends or restricts the formats of objects in the data space described by its modificand. For example, suppose we have defined a BinaryFormula as

\[
\text{BinaryFormula} \leftarrow \left[ \text{Operator:BinaryOperator} | \text{LeftOperand:Formula} | \text{RightOperand:Formula} \right]
\]

Then a LambdaFormula (representing a \( \lambda \)-expression) could be defined as follows:

\[
\text{LambdaFormula} \leftarrow \text{BinaryFormula} \Theta \ (\text{Operator} = (=\text{'lambda'})) \land \\
(\text{LeftOperand} = \text{List(\text{identifier})})
\]
The binary operator \( \Theta \) is pronounced 'with'. Here the descriptor for a LambdaFormula is a modified descriptor consisting of a modificand, which, in this case, is the elementary descriptor assigned as the value of the descriptor variable BinaryFormula above, and a modifier, which is the construction \( \Theta (\text{Operator} = (=\text{\em lambda})) \land (\text{LeftOperand} = \text{List(identifier)}) \). In the format modifier, following the \( \Theta \) operator, is a conjunction of equations of the form \((n_1=dss_1) \land (n_2=dss_2) \land \ldots \land (n_k=dss_k)\), for \( k \geq 1 \). Each of the \( n_i \), for \( 1 \leq i \leq k \), is either a component name or an expression whose value is a selection path (see Section 15), and each of the \( dss_i \), for \( 1 \leq i \leq k \), is a data space specifier. Thus the generic form of a modified descriptor incorporating a format modifier is

\[
d \Theta (n_1=dss_1) \land (n_2=dss_2) \land \ldots \land (n_k=dss_k)
\]

where \( d \) is an expression having an elementary descriptor as value. Let \( \text{val}(d) \) stand for the elementary descriptor which is the value of \( d \). The effect of this kind of modification is to execute the following process for each \( n_i \) in order of increasing \( i \) \( (1 \leq i \leq k) \). If \( n_i \) is a component name a linear search is made among the names of successive named component descriptions of \( \text{val}(d) \) (if there are any) for occurrences of \( n_i \) and upon locating in \( \text{val}(d) \) a component

\[\text{---}\]

† This effect is carried out at the time the value of a format modified descriptor expression is obtained.
description of the form \( \bar{n}: \text{dss} \) with \( \bar{n} = n_i \), \( \text{dss}_i \) is substituted for \( \text{dss} \), altering the component description to read \( \bar{n}: \text{dss}_i \). If no component of the form \( \bar{n}: \text{dss} \) can be found with \( \bar{n} = n_i \) then the component description \([n_i: \text{dss}_i]\) is concatenated to the end of \( \text{val(d)} \).

If \( \pi \) is a selection path, then the path is applied to \( \text{val(d)} \) to designate a (possibly nested) component description, and this component description is replaced by \( \text{dss}_i \). If the application of the selection path to \( \text{val(d)} \) results in no selection, a component description of the form \( \text{dss}_i \) is added at the appropriate level of nesting. (See section 15 for the meaning of application of selection paths to objects.)

The process terminates with the processing of \( n_k \).

In the example above (of the LambdaFormula) we begin by processing the first conjunct of (Operator\(=\)='lambda')\(\wedge\) (LeftOperand\(=\)List(identifier)). Thus, given that the value of the descriptor variable BinaryFormula is the elementary descriptor \([\text{Operator}: \text{BinaryOperator} | \text{LeftOperand}: \text{Formula} | \text{RightOperand}: \text{Formula}]\) we search linearly (ignoring any nested structures) among its named component descriptions, of which there are three, for one whose name is 'Operator', of which there is one, namely 'Operator:BinaryOperator', and we alter its data space specifier, namely 'BinaryOperator', to the data space specifier
found on the right hand side of the equation \( \text{Operator} = (=\text{lambda}) \).

This produces the modified descriptor

\[
\text{[Operator: (='lambda') | LeftOperand:Formula | RightOperand:Formula].}
\]

The second conjunct \( (\text{LeftOperand}=\text{List(identifier)}) \) is now processed and further modifies the descriptor to its final value

\[
\text{[Operator: (=\text{lambda}) | LeftOperand:List(identifier) | RightOperand:Formula].}
\]

Thus the effect of executing the assignment \( \text{LambdaFormula} \leftarrow \text{BinaryFormula} \Theta (\text{Operator}= (=\text{lambda})) \land (\text{LeftOperand} = \text{List(identifier)}) \)

is the same as executing the assignment \( \text{LambdaFormula} \leftarrow \text{[Operator: (=\text{lambda}) | LeftOperand:List(identifier) | RightOperand:Formula].} \)

Format modifiers are useful when defining several types of data structures that consist of minor variations on one basic data structure. They permit the class of slightly varying data to be defined without repeating large portions of the basic definition in each definition. The following example demonstrates this principle.

Suppose we have defined a SimpleFile by means of the statement

\[
\text{SimpleFile} \leftarrow \text{[name:indefiniteXcharacter | coordinates:[x:any | y:any] | type: string | marker: Boolean ]}
\]

Then we wish to define three different varieties of SimpleFile depending
on the type of their coordinates. This can be done as follows:

RealSimpleFile ← SimpleFile Θ (type=(='real'))^([2, 1]=real)^([2, 2]=real) ;
IntegerSimpleFile ← SimpleFile Θ (type=(='integer'))^([2, 1]=integer) ^ ([2, 2]=integer) ;
ComplexSimpleFile ← SimpleFile Θ (type=(='complex'))^([2, 1]=complex)^([2, 2]=complex) ;

The effect of executing these statements is the same as that of executing the following equivalent set of statements:

RealSimpleFile ←[name: indefinite X character | coordinates:[x:real | y:real] | type: (='real') | marker: Boolean ] ;
IntegerSimpleFile ←[name: indefinite X character | coordinates:[x:integer | y:integer] | type: (='integer') | marker: Boolean ] ;

By comparing the two equivalent sets of statements we see that the selector path [2, 1] locates for replacement the first component description of the descriptor which is the second component description of the descriptor for SimpleFile. (The [2, 1] component is the same as the x component of the coordinates component.)

Format modifiers have been found useful in practice for eliminating repetition in the definition of descriptors.
b. The Predicate Modifier

The purpose of a predicate modifier is to describe an extension to the predicate associated with a descriptor. The binary operator \( \supset \) (pronounced 'such that') is used to associate as modifier a monadic, Boolean \( \lambda \)-expression of the form \( \{ \lambda (x) \ B(x) \} \) with a modificand \( d \) which is a descriptor valued expression. Thus the generic form of a predicate modified descriptor is

\[
d \supset \{ \lambda (x) \ B(x) \} .
\]

The predicate associated with a predicate modified descriptor of this form tests whether an object, \( y \), has the format described by \( d \), and, in addition, whether \( y \) satisfies the property given by the Boolean expression \( B(x) \), which property is ascertained by evaluating \( \{ \lambda (x) \ B(x) \} [y] \) for truth or falsehood. An example of a predicate modifier is given in the following modified descriptor describing lists of length two:

\[
\text{TwoList} \leftarrow \text{List(any)} \supset \{ \lambda (x) \ \text{Length}(x) = 2 \} \text{ where}
\]

the procedure \( \text{Length}(L) \), which computes the length of a list \( L \) is given as follows:

\[
\begin{align*}
\text{integer procedure \ Length(L) ; List(any) L ;} \\
\text{Length} & \leftarrow \text{if nullist(L) then 0 else } 1 + \text{Length(tail(L)) ;}
\end{align*}
\]
The data structure for lists is given and explained in Section 6.

c. The Constructor Modifier

Constructor modifiers describe extensions to the constructor associated with a descriptor. The binary operator $\Delta$ (pronounced 'transformed by') is used to associate as modifier a monadic $\lambda$-expression of the form $\{\lambda(x) \ C(x)\}$ with a modificand, $d$, which is a descriptor valued expression. The generic form of a constructor modified descriptor is thus

$$d \ \Delta \ \{\lambda(x) \ C(x)\}$$

The constructor associated with a descriptor of this form constructs from $n$ arguments a datum, $z$, of the format given by $d$ (see Section 9) and it then applies the operator $\{\lambda(x) \ C(x)\}$ to $z$ by evaluating the expression $\{\lambda(x) \ C(x)\}[z]$. This results in performing the constructions specified by executing $C(z)$, where $C(z)$ is any statement or procedure call. The modifier $\{\lambda(x) \ C(x)\}$ therefore specifies an addition to the constructor associated with $d$. An example of a constructor modifier is given by the following descriptors and procedures which describe $n$ component rings constructed from sequences of $n$ pairs each having data and link
components.

pair ← [datum: any | link: .pair ];

descriptor procedure Ring (n) ; integer n ;
Ring ← ( n × pair ) Δ {λ(x) LinkRing(n, x) } ;

procedure LinkRing(n, x) ; integer n ; Ring(n) x ;
begin integer i ;
for i ← 1 step 1 until n-1 do link(x[i]) ← x[i+1] ;
link(x[n]) ← x[1] ;
end ;

Here the body of the procedure Ring(n) computes a constructor
modified descriptor consisting of an iterated format of n pairs modified
by a conversion procedure for linking these pairs into a ring once
storage for the sequence of pairs has been allocated at construction time.
We forgo giving detailed examples of allocation and construction of Rings
at this point because they are dependent on properties of constructors
defined later in Section 9. This omission is remedied, however, by
numerous examples of constructions given and explained in Chapter IV.
We take the description of the iterated sequence of pairs transformed
by the conversion operator for linking as the description of the class of
n element rings. Another possibility for a class of descriptors that would
describe rings (or in general structures isomorphic to directed graphs)
is to compute a descriptor which is itself a ring (or graph) and to create
representations of such structures at allocate time using a 1-1 mapping
which copies the capacity and connectivity directly. This possibility
is not pursued in this dissertation since it is desired to keep all
descriptor formulae in the form of operator-operand trees permitting

† We note in this example that procedure declarations are permitted
anywhere in a block and need not occur only in the block head. In fact,
this property is true of all declarations (cf. Section 7.)
simplicity in the reduction algorithms that operate on them. Essentially these reduction algorithms are abstract formula manipulation algorithms (abstract formulae being those which do not obey the laws of real fields but rather obey laws defined by their evaluation algorithms). There is no loss of generality in this approach since predicate and constructor modifiers allow the description of directed graphs with a fair amount of ease.

d. Combinations of Modifiers

A descriptor formula may have modifiers applied to it not only singly but also in combination. For example, constructions of the form \(d \oplus (n = \text{dsn}) \Delta \{\lambda (x) \ C(x)\} \supseteq \{\lambda (y) \ B(y)\}\) are permissible.

6. Descriptor Formulae

A descriptor formula is either an elementary descriptor, a modified descriptor, or a disjunction of elementary and modified descriptors. In the case of a disjunction the disjuncts are permitted to bear names. An example of a descriptor formula which is a disjunction is the following:

```plaintext
descriptor procedure List(t) ; descriptor t ;
List \leftarrow \text{nullist}::[\text{nil}] \lor \text{pair}(t) :: [\text{head} : t \mid \text{tail} : \text{List}(t)] ;
```

If we obtain the value of the function designator \(\text{List}(`\text{integer}`)\), `\text{integer}` is substituted for \(t\) throughout the body of the procedure, and we get
nullist::[nil] ∨ pair(integer)::[head:integer|tail: List.(integer)]

Here the descriptor formula for a list of integer's is a disjunction of two named disjuncts. The first disjunct is 'nullist::[nil]''.

This states that a nullist is an element with a single component described by the descriptor 'nil'. ('nil' is not only an identity under concatenation and composition of descriptors as explained in Section 4, it is also a descriptor valued variable whose value is assumed to be given by the assignment nil ← ['nil']). The second named disjunct is 'pair(integer)::[head:integer|tail:List.(integer)]'.

This states that a pair of integer's is an elementary descriptor with two components: a head chosen from the data space of representations of integer's and a tail which is chosen from the data space of lists of integer's. Notice that the name of a disjunct may bear parameters, e.g. pair(t), the name being an implicit definition of a descriptor.

Notice also that the name of the object being defined, namely List.(t), may be used recursively as a data space specifier in the descriptor formula of its own definition. The dotted function designator signifies the name of the procedure and the undotted form List(t) signifies the call of the function (as in Formula Algol [49]). If the undotted form were used in the above definition of List(t) it would result in an infinite regress of recursive calls. The two named elementary descriptors
for the nullist and the pair of integer's are joined by the binary operator \( \lor \) denoting disjunction. This signifies that the two disjuncts are descriptions of the two alternate possible formats for a list of integer's.

The Character Set Abbreviation

Several applications require the definition of character sets as basic sub-data spaces used in the definition of others. The system defined above permits character sets to be defined by descriptors which are disjunctions of unit data spaces. For example:

\[
\text{BinaryOperator} \leftarrow \left[ ='+1' \right] \lor \left[ '='-1' \right] \lor \left[ '='X' \right] \lor \left[ '='/\right] \]
\]

(1)

Not only is this type of definition clumsy for long definitions but also character sets occur frequently enough in applications so that it becomes convenient to introduce the following equivalent abbreviation for the construction in (1):

\[
\text{BinaryOperator} \leftarrow \{ +, -, \times, / \}
\]

Specifically, \( \{ +, -, \times, / \} \) is an abbreviation for the descriptor formula \( \left[ ='+1' \right] \lor \left[ '='-1' \right] \lor \left[ '='X' \right] \lor \left[ '='/\right] \). In general, \( \{ c_1, c_2, \ldots, c_n \} \) is an abbreviation for \( \left[ '='c_1' \right] \lor \left[ '='c_2' \right] \lor \ldots \lor \left[ '='c_n' \right] \) where the \( c_i \) are character strings of any length not including commas.
Character sets may be used as disjuncts in descriptor formulae. For example:

\[
\text{CompassDirection} \leftarrow \{N, E, S, W\} \setminus \text{[integer]} \ni \{ \lambda (x) \; (x > 0) \wedge (x \leq 360) \}
\]

7. Declaration of Descriptor Variables and Descriptor Procedures

Variables and function designators which are to take on descriptor formulae as values must be declared of type `descriptor`. Unlike normal Algol declarations, however, declarations in the data definition facility need not occur at the head of a block, but may instead be distributed through it. The effect of a declaration made in a block is revoked upon block exit.

An important property of descriptor variables is that they may be used in descriptor expressions before values are assigned to them. By convention, the symbol 'undefined' is taken to be the value of a descriptor variable that has been declared but not used on the left hand side of an assignment statement. When the value of such an undefined descriptor variable is used in computing the values of a sequence of descriptor expressions, a chain of its uses is created. When a value is assigned to it, a reference to this value is inserted at all places indicated on this chain. (This is a standard chaining algorithm used in compilers to handle undefined forward references.)
The use of this chaining algorithm is necessary to permit the introduction of co-definitions, i.e. systems of definitions each of which uses all of the others, and in which no definition can be made independent of the others permitting it to be given first. As an example, the set of definitions for data structures of a subset of Algol text (cf. pp. 104 ff.) contains the co-definition pair:

```
  descriptor labelledstatement, statement ;
  labelledstatement ← [label: identifier | text: statement] ;
  . . .
  statement ← conditional ∨ goto ∨ assignment ∨ labelledstatement
                     ∨ block ∨ dummy :: [nil] ;
```

In this example, the use of the value of the descriptor variable `statement` is required in the definition of a `labelledstatement` before the value is created and assigned. Reversing the order of the two statements does not improve the situation, since the value of the variable `labelledstatement` would then be undefined. Using the chaining algorithm given above, when the descriptor for `labelledstatement` is created, the value of the variable `statement` is 'undefined', and this value is inserted in the descriptor assigned as the value of the variable `labelledstatement`. In addition, a reference to the location of the use of the value 'undefined' is inserted on a chain associated with the
descriptor variable statement. When the value of the descriptor variable statement is computed and assigned, references to this value are inserted at those locations given by its associated chain.

8. Predicates, Selectors, Constructors and Declarations

Once a descriptor formula has been created it may be used to provide (1) construction of objects in the data space it describes, (2) selection of parts of objects in the data space it describes, (3) testing of the type and format of objects in the data space it describes, and (4) declaration of variables and procedures which take on values from the data space it describes. Sections 9, 10, 11 and 12 describe these various classes of operations and present the notations used to evoke them.

9. Constructors

Constructors are obtained by applying the functional named cons (short for construct) to descriptor valued expressions. Cons maps
descriptors into constructor functions which allocate storage and construct the datum described by the descriptor when applied to lists of parameters giving the parts to be used in the act of construction. For example, suppose the assignment
complex ← [realpart: real | imagpart: real] has been executed.
Then cons(complex) is a constructor function of two variables which constructs from two representations of real numbers a representation of a complex variable in the background machine. Thus cons(complex)(3.5, 2.0) constructs (when executed) a complex datum with realpart = 3.5 and imagpart = 2.0.

If the value of d, in cons(d), is a disjunction then cons(d) constructs the representation of the first disjunct. I.e.

\[ \text{cons} \left( a_1 \lor a_2 \lor \ldots \lor a_n \right) = \text{cons}(a_1) \, . \]

For example, using the definition of List(t) given on page 48,

\[ \text{cons}(\text{List}(x)) = \text{cons}(\text{nullist}::[\text{nil}] \lor \text{pair}(x)::[\text{head}:x|\text{tail}:\text{list}.(x)]) = \text{cons}(\text{nullist}::[\text{nil}]) = \text{cons}([\text{nil}]) . \]

This construction policy may be altered by an initial value construction given in section 12.
The naming of disjuncts in a descriptor formula causes implicit assignments of those disjuncts to be the values of those names. These names may be used as parameters to the functional \texttt{cons}, (e.g. \texttt{cons(nullist)} and \texttt{cons(pair(x))}) to signify the constructor functions for the named objects.

\textbf{Parameter Conventions}

When the descriptor \texttt{d}, in \texttt{cons(d)}, is an elementary descriptor with nested component descriptions, then the parameter list to which the function \texttt{cons(d)} is applied may be a list structure to specify the parts of nested components. For example, if \texttt{d} is the descriptor \texttt{[integer}[\texttt{integer}][\texttt{integer}]|\texttt{Boolean]} then \texttt{cons(d)(2, (3, 4), true)} specifies the construction of a data structure whose second component has the form \texttt{cons([integer][integer])(3, 4)}. Constructors allocate space according to the format given by \texttt{d} and expect to find values to assign to the cell structure of the allocated space in the parameter list to which they are applied.

In the case of the replicator \texttt{indefiniteX}[\texttt{C}], \texttt{cons(indefiniteX}[\texttt{C}]) finds the number of replications from the length of the parameter list to which it is applied and allocates the proper amount of space. For example, \texttt{cons(indefiniteX}[\texttt{real}])\texttt{)(3.5, 4.1)} allocates space for a pair of reals whereas \texttt{cons(indefinite X [real])(2.5, 3.6, 7.0)} allocates
space for a triplet of reals.

In the case of descriptors with component descriptions specifying unit data spaces, each data space specifier defining a unit data space is used to specify the datum used in the construction of its corresponding component. For example, if \( d = [\text{realpart:real} \mid \text{imagpart: (=0)}] \) then \( \text{cons}(d)(x,y) \) will construct a data structure whose second component is 0 no matter the value of \( y \). In this case, \( y \) may be omitted, and \( \text{cons}(d)(x) \) will have the same effect as \( \text{cons}(d)(x,y) \).

Parameters may also be omitted from \( \text{cons} \) by leaving them blank and the corresponding component of the constructed data structure will be left undefined (even though space for it will have been allocated). For example, if \( d = [\text{integer} \mid \text{integer} \mid \text{integer}] \) then \( \text{cons}(d)(2,_,4) \) constructs a triplet of integers whose second component is undefined, and \( \text{cons}(d)(2,,) \) constructs a triplet of integers whose second and third components are undefined, whereas \( \text{cons}(d) \) constructs a triplet of integers all of whose components are undefined. The undefined components may later be defined by assignments (see section 10).

The Effect of Cons on Modified Descriptors

When \( \text{cons} \) is applied to a format modified descriptor of the form \( d \oplus m \) the format modification is carried out first and the
resulting constructor function constructs the modified format.

When \( \text{cons} \) is applied to a predicate modified descriptor
of the form \( d \supseteq \{ \lambda (x) \ B(x) \} \) the predicate modifier is ignored.
I.e. \( \text{cons}(d \supseteq \{ \lambda (x) \ B(x) \}) = \text{cons}(d) \). When \( \text{cons} \) is applied
to a constructor modified descriptor of the form \( d \Delta \{ \lambda (x) \ C(x) \} \),
the operator given in the constructor modifier is applied to the
result of construction. I.e.

\[
\text{cons}(d \Delta \{ \lambda (x) \ C(x) \})(y_1, y_2, \ldots, y_n) = \\
\{ \lambda (x) \ C(x) \}[\text{cons}(d)(y_1, y_2, \ldots, y_n)] = \\
C(\text{cons}(d)(y_1, y_2, \ldots, y_n))
\]

**Construction of Tagged Data**

When a descriptor or descriptor valued expression, \( d \),
is prefixed by a unary double quote mark "d (e.g. "complex,
"List, (integer)) then \( \text{cons}(''d) \) is a function which constructs
a data structure identical to that constructed by \( \text{cons}(d) \) except
that it bears a tag which marks it of type \( d \) (which is a reference to
the descriptor \( d \)). A class of predicates is available to test for
the presence and type of this tag. Sometimes it is cheaper to
test for a tag than it is to test for well formedness of an entire data
format. This is especially true of data which is a connected graph
for which the generation algorithms may be difficult to write (because of the problem of avoiding generating loops indefinitely). We shall see later, in section 11, that ways exist to construct data bearing more than one tag simultaneously.

10. Selectors

Either component names or integers denoting the ordinal position of a component description within an elementary descriptor may be used to select components from a datum. For example, if \( x \) names the datum which results from obtaining the value of the expression \( \text{cons(complex)}(3.5, 2) \) in some background machine, then \( \text{realpart}(x) = 3.5 \) and \( \text{imagpart}(x) = 2 \). Instead of using the component names 'realpart' and 'imagpart' in the selection expressions 'realpart(x)' and 'imagpart(x)' we can use the integers 1 and 2 respectively in the expressions \( x[1] \) and \( x[2] \). Thus, \( x[1] = \text{realpart}(x) = 3.5 \) and \( x[2] = \text{imagpart}(x) = 2 \). The integers 1 and 2 are the ordinal positions of the realpart and imagpart component descriptions within the elementary descriptor \([\text{realpart:real}|\text{imagpart:real}] \). If the components are not named in an elementary descriptor, integer selectors must be used. These selection processes may be compounded in expressions of the following type:
(a) matrix(A)[3] finds 3rd component of the element obtained by selecting the 'matrix' component of A. Note that selection by component name takes precedence over ordinal selection by integer.

(b) valence(bonds(X)[4])

c) M[2][4][6]

d) M[2, 4, 6]

e) matrix(numerator(x[2]))[3, 5]

Examples (c) and (d) above have the same value because integer selectors may be given either by subscript list or by repeatedly appending expressions of the form [integer] to the end of a selection expression.

Selection expressions may be used on either the right hand side or the left hand side of assignment statements where, following the terminology of Strachey [62], they take on right hand values or left hand values respectively. Right hand values are components selected from a datum, and thus are data themselves, and left hand values are locations of components where assignments of data are to take place. Thus

\[ \text{valence (bonds(x)[3]) } \leftarrow 2 \times \text{valence(c[3])} \]

assigns twice the value of the valence component of the element that
is the third component of \( c \) to be the value of the valence component of the element that is the third component of the element that is the bonds component of the datum \( x \). Self referential assignments (e.g. \( X[3, 1] \leftarrow X[3] \)) are permitted and create self-referential structures unless a copy function is invoked [see Section 14].

11. Predicates

Let \( d \) be a descriptor valued expression and let \( x \) be an expression whose value is a datum. The expression \( x \leftarrow d \) is a predicate which is true if \( x \) is a datum whose representation in the background machine belongs to the data space of representations described by \( d \), and which is false otherwise.

If the value of \( d \) is an elementary descriptor (such as a nested format descriptor or a replicator) then \( x \leftarrow d \) tests whether \( x \) has the format described by \( d \) and whether the types of the components satisfy the component descriptions in \( d \). If the value of \( d \) is a disjunction of descriptor formulae then \( x \leftarrow d \) tests whether \( x \) is of the format given by at least one of the disjuncts.

If \( d \) is a format modified descriptor, \( x \leftarrow d \) will be true if \( x \) is an instance of the modified format given by \( d \). If \( d \) is a constructor modified descriptor of the form \( \tilde{d} \Delta \{ \lambda(x) \ C(x) \} \), the
constructor modifier will have no effect on the meaning of
\( x == d \) so that \( x == (\tilde{d} \Delta \lambda(x) C(x)) \) has the same truth
value as \( x == \tilde{d} \). If \( d \) is a predicate modified descriptor of
the form \( \tilde{d} \supseteq \lambda(y) B(y) \), the operator in the predicate modifier
must also be true when applied to \( x \) in order for \( x == d \) to be true
so that \( x == d \) is true iff both \( x == \tilde{d} \) and \( \lambda(y) B(y)[x] \) are
true. I.e.

\[
x == \tilde{d} \supseteq \lambda(y) B(y) = (x == \tilde{d}) \wedge \lambda(y) B(y)[x] =
(x == \tilde{d}) \wedge B(x).
\]

If a datum \( x \) has been constructed as tagged data by a
constructor of the form \( \text{cons}(''d) \), the presence of the tag may
be detected by a predicate of the form \( x == ''d \), which will be
true iff \( d \) is one of the tags associated with \( x \).

More than one tag may be associated with a constructed
datum simultaneously. For example, if we write the assignments

\[
\text{BinaryFormula} \leftarrow \text{[Operator:BinOp|LeftOperand:Formula |} \newline
\text{RightOperand: Formula] ;}
\]

\[
\text{LambdaFormula} \leftarrow ' '\text{BinaryFormula} \oplus (\text{Operator} = (''\text{lambda'}) \wedge \newline
(\text{LeftOperand} = \text{List.(identifier)}) ;
\]

Then if \( x == \text{cons}(''\text{LambdaFormula})(, \text{cons(pair(any))}(y, \text{cons(nullist)}), z) \)
it is the case that both $x \equiv \text{"LambdaFormula}$ and $x \equiv \text{"BinaryFormula}$ are true, whereas if "BinaryFormula had been replaced by BinaryFormula in equation (1) above, then $x \equiv \text{"BinaryFormula}$ would have been false. Thus if a quoted descriptor is used as a modificand in a modified format descriptor the associated constructor constructs data tagged with the quoted modificand. This reveals another use of format modifiers.

A Synonymous Notation for Predicates

The notation $x \equiv d$ may be replaced by $d(x)$ provided $d$ is a descriptor valued variable or function designator. E.g. complex$(x)$ is true iff $x \equiv \text{complex}$ and List(integer)$(x)$ is true iff $x \equiv \text{List(integer)}$.

Testing for Types

The function $\text{type}(x, d)$ is a special function which is integer valued and takes both a datum $x$ and a descriptor valued expression $d$ as arguments. The integer which is the result is the ordinal position of the first disjunct $\tilde{d}$ in $d$ which has the property that $x \equiv \tilde{d}$. If no such disjunct exists, then the value is zero. For example, given that $d \leftarrow \text{List(integer)}$, making the value of $d = \text{nullist} : : [\text{nil}] \lor \text{pair(integer)} : : [\text{head:integer} | \text{tail:List.(integer)}]$ the value of $\text{type(cons(pair(integer)(3,nil),d)}$ is 2, the value of
type ( cons(nullist), d ) is 1 and the value of type( cons(complex), d) is zero.

Use of Disjunct Names in Predicates

The names of named disjuncts in descriptor formulae may also be used in forming predicates. Recall (Section 6) that names of disjuncts are implicitly assigned the disjuncts they name as values within a descriptor formula and that they therefore enjoy all of the properties of descriptor variables. For example, nullist(x), x == nullist, pair(integer)(x) and x == pair(integer) are all perfectly good predicates. Likewise cons(nullist) and cons(pair(integer)) are perfectly good constructors.

The Recursive Equality Predicate

Once a descriptor has been defined or computed and once it has been used to construct data, the constructed data may be tested for equality automatically (provided they are not self-referential). Let S be a set of data of type d where the formats in d are nested or iterative, and let x and y belong to S. Then x = y is a well defined equality test among members of S which will be true if x and y have the same format and have equal components, and will be false otherwise. The equality of components is tested recursively.
12. Declarations

Let $d$ be a descriptor valued variable or function designator. Then $d$ may be used as a declarator to declare variables which may take on values from the data space of representations associated in the background machine with the descriptor formula which is the value of $d$. For example,

\begin{verbatim}
complex U, V, Z, W;
List(integer) L, M;
matrix(m, n, complex) K, F, G;
\end{verbatim}

**Initial Value Policy**

If it is desired to give initial values to the declared variables this may be done within the body of a descriptor procedure by use of an *initial value* assignment. For example:

\begin{verbatim}
descriptor procedure formula;
begin
  formula ← unaryformula ∨ binaryformula ∨ atom;
  initial value ← declarand;
end;
\end{verbatim}

Here the statement 'initial value ← declarand' causes the initial value of the declared variable $V$ to be the name $V$ in the declaration
'formula V '. Variables declared in the identifier list of the declaration are declarands. In the process of declaration the declarator is defined to distribute over the declarands and to process them one by one. E.g. 'formula U, V, W ;' is the same as and expands to 'formula U ; formula V ; formula W ;'.

Another example is:

```plaintext
descriptor procedure complex;
begin
complex ← [realpart: real | imagpart: real];
initial value ← cons(complex)(0, 0);
end;
```

Here the declaration 'complex U, V, Z, W ;' initializes the value of the declarands U, V, Z and W to be cons(complex)(0, 0).

Declaring Variables of Arbitrary Type

The type word _any_ may be used to declare variables whose value may be data of any type whatsoever. For example, _any x, y, z_; declares x, y and z to be variables which can take on data constructed with any constructor definable in the system.

13. Reference Variables, Pointer Expressions and The Contents Operation
Variables or procedures declared of type reference may take on as values pointers or references to places within allocated data formats. References may be used in the construction of data structures and are computed by means of pointer expressions. A pointer expression is a selection expression prefixed by the unary operator '→'. For example:

\[ \rightarrow \text{realpart}(x) \]

\[ \rightarrow \text{valence}((\text{bonds}(x))[4]) \]

The reference operator '→' causes the left hand value of E in an expression of the form \( \rightarrow E \) to become the value of the expression. The contents operation, signified \( <E> \), is the inverse operation of the reference operation \( \rightarrow E \). Given a left hand value, the contents operation obtains the corresponding right hand value. The identity \( <\rightarrow E>E \) demonstrates the inverse property of the contents and reference operators.

The existence of references permits the construction of richly connected data and data with loops (i.e., data in the form of directed graphs). For example, the following descriptor describes a class of data which are triplets having the property that the first component is a reference to the third component.
M ← [ a: reference | b: real | c: real ] \{ \lambda (x) \, x[1] \leftarrow (\neg x[3]) \}

The modifier \{ \lambda (x) \, x[1] \leftarrow (\neg x[3]) \} is an operator which assigns
a reference to the third component of its operand (\neg x[3]) to be
the value of the first component of its operand (x[1]).
Given
the datum \( c \leftarrow \text{cons}(M)(, 3.5, 2.0) \) the value of \( c[1] \) is a
reference to \( c[3] \), the value of \( c[3] \) is 2.0 and the value of
\(< c[1] > \) is 2.0.

Note that the value of \( \neg x[3] \) is quite different from the
value of \( x[3] \). Had the constructor modifier above been replaced
by \{ \lambda (x) \, x[1] \leftarrow x[3] \} instead of \{ \lambda (x) \, x[1] \leftarrow (\neg x[3]) \} then the
constructor \text{cons}(M) \ would have constructed triplets whose first
component referred to the same datum as referred to by the third
component. The first and third components would then share a
reference to the value of \( x \) in \text{cons}(M)(, y, x). This is different from
the former case, introduced above, in which the first component was
a reference to the location within the format \text{cons}(M) \ of the third
component (where a reference to \( x \) was stored). This is a subtle
difference. It amounts to saying that in formats of nested structure
the components consist of references either to atomic data or to
nested sub-formats, and that all reference variables add is the
capability to refer to the locations where references to substructures
are stored, these locations being locations within allocated formats. By compounding pointer expressions, arbitrarily many levels of indirectness may be created.

14. Overlay Assignments, Sharing of Structures and Copying of Structures

In some applications two or more structures may want to arrange to share a common substructure. For example, in a medical file system in a hospital, access to parts of a patient file may need to be shared both by a doctor, by the kitchen, and by the laboratory. The doctor should be able to request certain diets to be prepared by the kitchen, and certain diagnostics to be performed by the laboratory, and the kitchen and laboratory should have access to these requests stored as part of a patient file. (More elegantly a doctor should be able to write a program to inspect the results of laboratory diagnostics and to update the patients diet [as to, say, salt content] as a result, and the kitchen should have a program to inspect the files of all patients to prepare a summary of the kinds of meals to be cooked and the amount of supplies needed for each). This kind of system could be arranged, for example, by having the kitchen file share diet related portions of
patient files. Pointer expressions allow references to constant locations within other structures where pointers to shared structures are to be found. These shared structures may vary but the indirect addressing provided by pointer expressions provides access. Another type of sharing and updating can be done without indirect addressing. This is accomplished by the overlay assignment, of the form \( x \leftarrow E \). This causes the right hand value of \( E \) (a datum) to be stored at the location of the previous value of \( x \), so any shared references to \( x \) will now be shared references to the right hand value of \( E \). (In background machines where fixed blocks of storage are used to represent allocations of space for data formats and where the value of \( E \) requires too much space to overlay (block transfer) into the space previously occupied by \( x \), overlay assignments can be implemented by scanning addresses in memory and changing all references to the block occupied by \( x \) to a block allocated for \( E \).)

Finally, in the event that sharing of structures is to be prohibited, a function \( \text{copy}(E) \) exists which produces, non-recursively for nested formats, a separate copy of the right hand value of \( E \). The function \( \text{copyall}(E) \) copies all nested formats of the right hand value of \( E \) but cannot be used on data with self referential loops. This \text{copyall} function copies nested formats recursively.
15. Paths, Path Variables and Path Concatenation

Variables and function designators declared of type \texttt{path} may take on as values access paths which are, in essence, subscript lists and which are used in a succession of ordinal selections to select components of nested formats. Access paths may be concatenated by use of the concatenation operator '\_\_\_' as in the expression \texttt{x \_\_\_ y} where \texttt{x} and \texttt{y} are path valued expressions. They may further be applied to data using selection expressions of the form \texttt{x[p]} where \texttt{p} is a path valued expression. If \texttt{d} is a descriptor which is a subformat of the descriptor \texttt{D}, then the value of the function \texttt{path(d, D)} is a subscript list (\texttt{path}) indicating the path to the first instance of \texttt{d} in \texttt{D} (taken in sequence element mode search, which dictates that all left descendants be searched in \texttt{D} before any right siblings). For example, let the following declarations and assignments take place:

\begin{verbatim}
    path p ; descriptor complex, cpair, example ;
    complex ← [realpart:real | imagpart:real ] ;
    cpair ← [complex  | complex ] ;
    example ← [integer  | cpair  | Boolean ] ;
    p ← path ( complex, example ) ;
\end{verbatim}

The value of \texttt{path (complex, example)} is the subscript list \([2, 1]\).
This list is assigned to be the value of p. Thus \( x[p] \) computes the same result as the selection expression \( x[2, 1] \). The value of the expression \( \text{path}(\text{real}, \text{example}) \) is the subscript list \([2, 1, 1]\). Hence the value of \( p \cap \text{path}(\text{real}, \text{example}) \) is the subscript list \([2, 1, 2, 1, 1]\) which results from the concatenation of \([2, 1]\) and \([2, 1, 1]\). The value of \( p \cap \text{path}(\text{real}, \text{example}) \) could be used, in turn, in selection expressions such as \( x[p \cap \text{path}(\text{real}, \text{example})] \).

Paths may be derived by other kinds of operations on descriptors such as those that can be synthesized from the predicates, selectors, and constructors mentioned in Section 16.

16. Other Operations on Descriptors

Descriptor formulae, of course, have a data structure of their own. This data structure is given in Appendix III. Any of the predicates, selectors and constructors that result from the definition of the data structures of descriptors may be used as operations on descriptors. This immediate advantage results from having a system which is capable of defining itself, as is the case with descriptors.
17. Parallel Assignments and Block Expressions

Two convenient devices, originally given in the design of CPL [62], are used in the sequel. The first is the parallel assignment, denoted by expressions of the form \( x_1, x_2, \ldots, x_n \leftarrow e_1, e_2, \ldots, e_n \). This signifies that the assignments \( x_1 \leftarrow e_1, x_2 \leftarrow e_2, \ldots, x_n \leftarrow e_n \) are to take place in parallel (at the same time). The method of implementation in a sequential machine is first to compute and save all left hand values of \( x_1, x_2, \ldots, x_n \), second to compute and save all right hand values \( e_1, e_2, \ldots, e_n \), and third to make assignments of right hand values to their corresponding left hand values. Parallel assignments save the necessity of declaring and using temporary storage in sequences of assignments which switch values of variables and they are a convenient abbreviation.

The second device is the block expression whereby the value of a block is specified by assigning a value to the special variable \texttt{result} somewhere between the \texttt{begin end} block delimiters. For example, the block

\begin{verbatim}
begin integer i; sum ← 0; for i ← 1 step 1 until n do
    sum ← sum + A[i]; result ← sum; end;
\end{verbatim}

has a value which is equivalent to \( \sum_{i=1}^{n} A[i] \), where \texttt{sum}, \texttt{n}, and \texttt{A[i]} are global to the above block.

† Prof. D. Parnas has pointed out to me that the word parallel is not well defined, in this case, unless we agree that all expressions \( e_i \) are to be evaluated before any assignments are made.
Chapter IV

Examples and Applications

Purpose

The examples and applications in this chapter are intended to explore the use of the data definition facility as a synthetic tool for describing data and processes in several domains. In making this exploration, it is hoped that several principles will be shown, among which are: (1) that there is value in making the definitions of data structures explicit, in particular, conciseness of description, clarity of structure and neatness of organization are attained in the definition of data spaces and processes, (2) that the data definition facility can be used at the full level of complexity and sophistication demanded for implementing languages of practical utility (such as Lisp and Formula Algol), and (3) that adding a data definition facility to a language significantly improves its versatility and generality with regard to the number of data spaces with which it can deal gracefully.

1. Formula Manipulation

The data of interest in formula manipulation are the set of algebraic formulae to be used in such tasks as the non-numerical solution of algebraic equations, the differentiation and integration of formulae, the simplification and expansion of formulae, and other problems in formal algebraic manipulation, (see reference [56] for further examples, explanation and references).

The method of representing algebraic expressions can be chosen in a number of ways depending on the intended
applications of the system in which they are embedded. For example, the ALPAK system [12], which deals primarily with polynomials, rational functions, and matrices of rational functions, represents polynomials as arrays of coefficients, the FORMAC system [9] represents algebraic expressions internally as polish delimiter chains, and the SAINT system [60] uses Lisp list structures to represent expressions that are integrands to an integration routine. Formula Algol [49] uses binary trees to represent algebraic expressions, and this representation has been found moderately effective in practice by Iturriaga [29], the main drawbacks being that the processing of polynomials where it is desired to perform such operations as collection of like terms is made somewhat less efficient when compared with systems which use coefficient arrays or n-ary operators, and that the associative structure of a formula must always be taken into account when processing it. Several pleasant features of the Formula Algol system include the availability of a set of basic operations from which formula manipulation algorithms may be written compactly and with clarity of structure not only over the class of formulae that obey the axioms of real fields but over other classes of formulae as well (e.g. Boolean formulae, formulae representing elements of groups and rings, and formulae representing data structure descriptions [see Chapter VI]). This set of basic
operations includes constructor functions for algebraic formulae (evoked by writing the formula as an Algol arithmetic expression), a set of pattern matching operations whereby a given formula $F$ can be compared with a set of formula patterns $P$ to determine if the structure of $F$ is described by the members of $P$, an evaluation operator $\text{eval}(F)$ which carries out arithmetic on constants and performs several categories of formula simplification, a substitution operation which substitutes expressions for occurrences of atomic formulae in a given formula, a replacement operation in which a subexpression of a formula $F$ matching a pattern $P$ is extracted and replaced with an atomic formula $A$, and a Markov Algorithm operation in which a set of reductions to be performed on a formula (such as the reductions used to clear a fraction, or to differentiate or expand a formula) may be expressed in tabular form. These basic operations are adequate to express, with some elegance, a wide class of formula manipulation algorithms (see [29]). It is the purpose of this example to show that these operations may be described with brevity by means of the data definition facility, and thus that the structure of a formula manipulator may be reduced to a compact set of routines suitable for embedding in other systems of more general scope and intent at low cost.
Data Structures

The data that we will define with a set of formal data structure definitions will be the set of binary formulae and unary formulae representing arithmetic and Boolean expressions in Algol. Procedure calls, array expressions and lambda expressions will also be represented. For example, the binary tree expression for the arithmetic expression

\[
a + b \times (c - 3)
\]

is

\[a + b \times (c - 3)\]

The trigonometric, inverse trigonometric, logarithmic, and exponential functions are represented by unary operators: e.g.

\[
sin(x) / cos(x)
\]

is represented as

\[/\sin \cos/\]

x x

The procedure call \(P(x, y)\) is represented as

\[/P/\]

\[x x y\]

\[P\]

\[x x y\]

\[nil\]

and the array expression \(A[i, j]\) is represented as

\[/\]

\[A\]

\[i\]

\[j\]

\[nil\]
The following set of data structure definitions defines the set of unary and binary formulae.

**descriptor** Atom, Formula, UnaryFormula, BinaryFormula, UnaryOperator, BinaryOperator, ArrayFormula, ProcedureFormula, LambdaFormula, ProcedureOperator, PatternPrimary;

Atom ← [real] ∨ [integer] ∨ [Boolean] ∨ [string] ∨ [identifier] ∨ PatternPrimary;

UnaryFormula ← [Operator:UnaryOperator | Operand:Formula];

BinaryFormula ← [Operator:BinaryOperator | LeftOperand:Formula | RightOperand:Formula];

Formula ← UnaryFormula ∨ BinaryFormula ∨ Atom ∨ List(Formula) ∨ 
            LambdaFormula ∨ ProcedureFormula ∨ ArrayFormula;

UnaryOperator ← {-,-,., sin, cos, tan, ln, exp, arctan, =, =, =, of};

BinaryOperator ← {+,-,×,/, ↑,.;=, =, [, (, ∨, ∧, →, <, >, λ} ∨ [=', '];

PatternPrimary ← {real, integer, Boolean, formula, atom, list, any};

ArrayFormula ← 'BinaryFormula ⊙ (Operator={}) ∧ (LeftOperand =identifier) ∧ 
                           (RightOperand=List(Formula))

ProcedureFormula ← 'BinaryFormula ⊙ (Operator={}) ∧ 
                          (LeftOperand =ProcedureOperator) ∧ (RightOperand = 
                          List(Formula));

LambdaFormula ← 'BinaryFormula ⊙ (Operator={λ}) ∧ (LeftOperand =
                          List(identifier));

ProcedureOperator ← identifier ∨ LambdaFormula ∨ ArrayFormula ∨ 
                      UnaryOperator ∨ BinaryOperator;
descriptor procedure List(t) ; descriptor t ;

    List ← Nullist :: [nil] ∧ Pair(t) :: ['BinaryFormula ⊕
    (Operator = ('=', '!')) ∧ (LeftOperand = t) ∧ (RightOperand = List(t)) ] ;

Synonyms

    Formula Procedure BoundVariableList(x);
    BoundVariableList ← LeftOperand(x) ;

    Formula Procedure Tail(x) ; Tail ← RightOperand(x) ;

    Formula Procedure Head(x) ; Head ← LeftOperand(x) ;

Comments on Background Machine

    We observe that the primitive predicates x = real ,
    x = integer , x = string , x = identifier , and x = Boolean are
    already defined as the primitive type tests for data space specifiers.

    [ A synonymous notation is real(x) , integer(x) , string(x) , identifier(x) and
    Boolean(x) , (see Chapter III, Section 11 ) ] . We also observe that
    the recursive equality test x = y depends on a polymorphic test for
    atomic equality [symbolized in the data definition facility as x=y
    (see Chapter III, Section 11 ) ] and defined explicitly as follows:

    Boolean Procedure AtomicEqual(x, y); any x, y ;
    AtomicEqual ← if real(x) ∧ real(y) then realequal(x, y) else
    if integer(x) ∧ integer(y) then integerequal(x, y) else
    if string(x) ∧ string(y) then stringequal(x, y) else false ;

    where realequal(x, y) , integerequal(x, y) and stringequal(x, y) are
monomorphic equality tests given for real's, integer's and string's in the Algol (or other) background machine.

Boolean values will also be represented by two strings 'true' and 'false' which are the values of the reserved string variables true and false respectively. Thus, the function \(\neg x\) can be represented by 'if \( x = \text{true} \) then false else true' which is background machine equivalent to 'if stringequal(x,'true') then 'false' else 'true' '.

Three functions used in the implementation of the current Formula Algol system [49] will be used in the sequel. The first function is EnvironmentAssign(x,E) which assigns the value of the expression E to be the value of the current binding of the identifier x in the current block and/or recursion level. The second function is Call(P,L) which calls procedure P with list of parameters L, and the third function is ArrayAccess(A,L) which accesses the element of the array A indicated by the subscript list L. We have chosen not to give the definitions of these three functions because they involve an aspect of the evaluation process separate from the one of current interest (which aspect deals with such issues as pertain to environment data structures and Algol models of procedure calls, block entry and exit, dynamic storage allocation and so forth.)
Procedures for Pattern Matching

The procedures given below for pattern matching make use of two types of tests called literal identity tests and pattern instance tests respectively. A literal identity test is a straightforward recursive equality test which checks if two formulae $F$ and $P$ are exactly identical operator for operator and atom for atom. Specifically, if $F$ and $P$ are both atoms then they are literally identical if the atoms are identical. If $F$ and $P$ are both binary formulae then they are literally identical if their operators are identical and if their left and right operands are literally identical respectively. If $F$ and $P$ are both unary formulae then they are literally identical if their operators are identical and if their operands are literally identical. In all other cases, $F$ and $P$ are not literally identical. For example, $(a + b \times c) / 2$ is literally identical to $(a + b \times c) / 2$ but is not literally identical to $(a + b \times c \uparrow 2) / 2$.

A pattern instance test compares a formula $F$ with a second formula $P$, called a pattern, defining a class of formulae with common structural features. $F$ is an instance of $P$ if it belongs to this class. For example, the pattern formula + formula matches any binary formula whose operator is '+' and whose right and left operands are formulae, the pattern $3 \times (\text{formula} - \text{real})$ matches any binary formula whose
operator is '×', whose left operand is the integer 3 and whose right operand is a difference of a formula and a real, and the pattern \( x : \text{real} - y : \text{formula} \) matches any formula \( F \) which is a difference of a real and a formula and it further extracts the left and right operands of \( F \) and assigns them as the values of the current bindings of \( x \) and \( y \) respectively.

Once a literal identity test and a pattern instance test are defined it becomes possible to define tests which are combinations of them, and, in fact, this ability becomes crucial when we desire to test the structure of patterns with other patterns (patterns being a species of formulae). For example, we might want to test a pattern \( K \) to see if its operator is '+', if its left operand is literally identical to 'formula - real', and if its right operand is a pattern instance of an integer. To do this, we could test whether \( K \) is a pattern instance of the pattern \( (= (\text{formula} - \text{real}) + \text{integer}) \) and we could arrange for the pattern instance routine to interpret the unary equal sign applied to the subexpression \((\text{formula} - \text{real})\) as an instruction to switch the mode of the test so that the left operand of \( K \) would be tested for literal identity against \( \text{formula} - \text{real} \) rather than being tested as a pattern instance against the corresponding left operand of the pattern \((= (\text{formula} - \text{real}))\). Thus testing a formula \( F \) as a pattern instance
of \( =P \) is equivalent to testing \( F \) for literal identity with \( P \). By symmetry, we could arrange for the literal identity routine to switch the mode of its test to pattern instance matching when it detects an expression preceded by the unary operator '\( == \)'. Thus testing \( F \) for literal identity with \( ==P \) would be equivalent to testing \( F \) to see if it is a pattern instance of \( P \). An equivalent way to find out if \( K \) is a pattern whose operator is '\( + \)', whose left operand is literally identical to 'formula - real' and whose right operand is a pattern instance of an integer is to test \( K \) for literal identity with the pattern \( \text{formula - real} + ( == \text{integer}) \).

In the routines given in the sequel \( \text{ExactlyEqual}(F, P) \) tests \( F \) for literal identity with \( P \), \( \text{Inst}(F, P) \) tests \( F \) to see if it is a pattern instance of \( P \) and \( \text{PrimitiveType}(x, y) \) is an auxiliary subroutine used by \( \text{Inst}(F, P) \). The routines \( \text{ExactlyEqual}(F, P) \) and \( \text{Inst}(F, P) \) are programmed as recursive co-routines allowing each to switch its mode of testing (from literal identity testing to pattern instance testing and vice versa) when indicated in the pattern \( P \). \( \text{PrimitiveType}(x, y) \) accepts an atomic datum \( x \) and a string \( y \) indicating a data type, which string is a PatternPrimary, and determines if \( x \) is of the type indicated by the string \( y \).

**Boolean Procedure** \( \text{PrimitiveType}(x, y) \); Formula \( x \); PatternPrimary \( y \);

\[
\text{PrimitiveType} \leftarrow \begin{cases} 
\text{if } y = '\text{real}' \text{ then } \text{real}(x) \text{ else } \\
\text{if } y = '\text{integer}' \text{ then } \text{integer}(x) \text{ else } \\
\text{if } y = '\text{string}' \text{ then } \text{string}(x) \text{ else } \\
\text{if } y = '\text{Boolean}' \text{ then } x = \text{true} \lor x = \text{false} \text{ else } \\
\text{if } y = '\text{list}' \text{ then } \text{BinaryFormula}(x) \lor \\
\text{Operator}(x) = ',', \text{ else } \\
\text{if } y = '\text{formula}' \text{ then } \text{Formula}(x) \text{ else } \\
\text{if } y = '\text{any}' \text{ then } \text{true} \lor \text{false} ;
\end{cases}
\]
Boolean Procedure ExactlyEqual(F, P); Formula F, P;

ExactlyEqual ←

if F == "BinaryFormula" ∧ P == "BinaryFormula" then

  ( Operator(F) = Operator(P) ) ∧
  ExactlyEqual(LeftOperand(F), LeftOperand(P)) ∧
  ExactlyEqual(RightOperand(F), RightOperand(P))

else

if F == "UnaryFormula" ∧ P == "UnaryFormula" then

  begin if Operator(P) = '==' then result ← Inst(F, Operand(P))
    else result ← (Operator(F) = Operator(P)) ∧
                 ExactlyEqual(Operand(F), Operand(P)) end

else

if Atom(F) ∧ Atom(P) then F = P else

if UnaryFormula(P) ∧ Operator(P) = '==' then Inst(F, Operand(P))

else false ;

Boolean Procedure Inst(F, P); Formula F, P;

if P == "UnaryFormula" then

  begin

    if Operator(P) = 'of' then begin Formula A ; A ← Operand(P) ;
      Inst ← eval(subst(Head(RightOperand(A), A, F)) end ;

    if Operator(P) = '=' then Inst ← ExactlyEqual(F, Operand(P)) ;

    if F == "UnaryFormula" then
      Inst ← (Operator(F)=Operator(P)) ∧ Inst(Operand(F), Operand(P))

  end
else

if LambdaExpression(F) ^ LambdaExpression(P) then
    begin  Formula B, C ;  B ← BoundVariableList(F) ;
            C ← BoundVariableList(P) ;
            Inst ← Inst(subst(B, LeftOperand(F), C), LeftOperand(P)) ;
    end

else

if P == 'BinaryFormula ^ Operator(P) = ':' then
    begin Boolean K;  K ← Inst(F, RightOperand(P)) ;
            if K then EnvironmentAssign(LeftOperand(P), F) ;
            Inst ← K ;
    end

else

if F == 'BinaryFormula ^ P == 'BinaryFormula then
    Inst ← (Operator(F) = Operator(P)) ^
            Inst(LeftOperand(F), LeftOperand(P)) ^
            Inst(RightOperand(F), RightOperand(P)) ;

else

if Atom(F) ^ Atom(P) then
    Inst ← if PatternPrimary(P) then PrimitiveType(F, P)
            else  F = P ;

else  Inst ← false ;

Pattern Containment

Once the pattern routine Inst(F, P) has been defined it becomes easy to write another routine Contains(F, P) which determines recursively whether the formula F contains a subexpression which is
an instance of P. \[ \text{Inst}(F, P) \] answers the question of whether F itself is an instance of P, not whether there exists some subexpression of F which is an instance of P \]. This procedure is defined as follows:

\[ \text{Boolean Procedure Contains (F, P); Formula F, P;} \]

\[ \text{Contains } \begin{cases} \text{if Inst}(F, P) \text{ then true else} \\ \text{if } F = \text{"BinaryFormula" then} \\ \text{Contains(LeftOperand(F), P)} \lor \\ \text{Contains(RightOperand(F), P)} \text{ else} \\ \text{if } F = \text{"UnaryFormula" then} \\ \text{Contains(Operand(F), P)} \text{ else false} \end{cases} ; \]

This operation is symbolized in Formula Algol [49] by the notation \( F \gg P \).

\textbf{Substitution}

The following substitution routine substitutes formula expressions for occurrences of atoms in a formula \( F \). \( L_1 \) and \( L_2 \) are assumed to be lists of equal length. \( L_1 \) is a list of atoms, and \( L_2 \) is a list of corresponding subexpressions to be substituted for occurrences of the atoms of \( L_1 \) in \( F \).

\[ \text{Formula Procedure subst(L1, F, L2); List(Formula) L1, L2; Formula F ;} \]

\[ \text{subst } \begin{cases} \text{if Atom(F) then} \\ \text{begin if } \sim \text{Nullist}(L1) \text{ then result } \begin{cases} \text{begin if Head(L1)= F then result } \sim \text{Head}(L2) \\ \text{else result } \sim \text{subst(Tail(L1), F, Tail(L2)) end} \end{cases} \\ \text{else result } \sim \text{F } \end{cases} \]
else
  if F == 'BinaryFormula' then
    cons('BinaryFormula')(Operator(F),
    subst(L1, LeftOperand(F), L2), subst(L1, RightOperand(F), L2))
  else
    if F == 'UnaryFormula' then
      cons('UnaryFormula')(Operator(F),
      subst(L1, Operand(F), L2));

Markov Algorithms

One of the more convenient methods of writing certain classes of formula transformations in Formula Algol is to use Markov Algorithms. Markov Algorithms permit one to express the essential transformations that constitute members of a larger transformation in tabular form. An example is the Markov Algorithm for clearing fractions. For simplicity, let G be a class of arithmetic expressions involving only binary +, binary × and binary / as operators. For each member of G it is desired to find an expression which is algebraically equivalent but which contains at most one occurrence of binary /. The following list of transformations can be used to express a fraction clearing transformation on G:

a: formula + b:formula / c:formula → (a×.c + b)/.c ,

a: formula / b:formula + c:formula → (a + b×.c)/.b ,

a: formula × (b:formula/c:formula) → (a×.b)/.c ,

(a: formula / b:formula) × c:formula → (.a×.c)/.b ,
This list consists of six transformations of the form \( P \rightarrow T \),
where \( P \) is a pattern containing extractor variables \( a, b, \) and \( c \),
and where \( T \) is a formula whose structure indicates how extracted
subexpressions of a matched formula are to be rearranged to
produce a transformed formula. Let \( E \) be an arithmetic expression
in \( G \). To apply the above list of transformations to \( E \) one tests
to see if \( E \) is an instance of the pattern \( P_1 \) on the left hand side of
the first transformation \( P_1 \rightarrow T_1 \) in the list of transformations.
If \( E \) is an instance of \( P_1 \) then the extractor variables \( a, b, \) and \( c \)
will have extracted as values matching subexpressions of \( E \), and this
permits one to substitute these extracted subexpressions for occurrences
of the atomic names \( .a, .b \) and \( .c \) in the expression \( T_1 \). If \( E \) does
not match \( P_1 \), control passes to the next transformation \( P_2 \rightarrow T_2 \) to
see if \( E \) matches \( P_2 \). If \( E \) is an instance of \( P_2 \) it is rearranged
according to the expression \( T_2 \), otherwise control continues in
sequence through the transformations until one matches or all fail.
If all transformations fail the entire sequence is reapplied to the
left and right operands of \( E \) recursively. If any transformation
applies, \( E \) is rearranged and control returns to the beginning of the
transformation sequence where the entire sequence is reapplied to
the transformed result. For example, if the above sequence of transformations is applied to the formula \( x + y/(u \times (v/w)) \) the first transformation,

\[
a: \text{formula} + b: \text{formula} / c: \text{formula} \rightarrow (. \ a \times . \ c + . \ b)/. \ c
\]

applies because the pattern \( a: \text{formula} + b: \text{formula} / c: \text{formula} \) matches \( x + y/(u \times (v/w)) \) in which the subexpression \( x \) is extracted into the variable \( a \), the subexpression \( y \) is extracted into the variable \( b \), and the subexpression \( u \times (v/w) \) us extracted into the variable \( c \).

This is accomplished by executing \( \text{Inst}(F, P) \) with \( F \) replaced by the formula and \( P \) replaced by the pattern (see definition if \( \text{Inst}(F, P) \) above). The formula is then rearranged by substituting \( x, y, \) and \( u \times (v/w) \) for \( . \ a, . \ b \) and \( . \ c \) respectively in \( (. \ a \times . \ c + . \ b)/. \ c \). This operation is signified by \( \text{subst}((. \ a, . \ b, . \ c), (. \ a, \times . \ c, . \ b)/. \ c, (x, y, u \times (v/w))) \), (see definition of \( \text{subst}(L1, F, L2) \) above). In the algorithm below we shall use the function \( \text{Replace}(F) \) to denote the replacement of all atomic variables in \( F \) by their current extracted values [see definition of Formula Algol [49]]. The result of this replacement is to rearrange the formula \( x + y/(u \times (v/w)) \) to the result \( (x \times (u \times (v/w)) + y)/(u \times (v/w)) \).

The sequence of transformations is then reapplied to this result. None of the transformations reapsplies to this result so the entire sequence is reapplied to the left operand \( (x \times (u \times (v/w)) + y) \) and to the
right operand \((u \times (v/w))\) recursively. Recursive application leads
to the use of the third transformation

\[
a: \text{formula} \times (b: \text{formula} / c: \text{formula}) \rightarrow (a \times . b) / . c
\]

to rearrange \(u \times (v/w)\) into the expression \((u \times v) / w\), and

iteration of the application of the whole sequence of transformations
to the rearranged result leads to the final series of rearrangements:

\[
x + y / (u \times (v/w)) \rightarrow (x \times (u \times (v/w)) + y) / (u \times (v/w)) \rightarrow
\]

\[
(x \times ((u \times v) / w) + y) / ((u \times v) / w) \rightarrow ((x \times u \times v) / w + y) / ((u \times v) / w) \rightarrow
\]

\[
((x \times u \times v + w \times y) / w) / ((u \times v) / w) \rightarrow (x \times u \times v + w \times y) / (w \times ((u \times v) / w) \rightarrow
\]

\[
(x \times u \times v + w \times y) / (w \times u \times v) / (x \times u \times v + w \times y) \times w) / (w \times u \times v).
\]

The last expression in the sequence is a cleared fraction. It could be
simplified by removal of common factors from the numerator and
denominator.

Let \(L\) be a list of formula transformations representing a
Markov Algorithm and let \(E\) be an expression to be transformed.
The following series of procedures defines the application of
\(L\) to \(E\) resulting in its transformation, where \(\text{Markov}(E, L)\) performs
the actual transformation.

\[
\text{Formula Procedure Markov}(E, L) ; \text{Formula } E, L ;
\]

\[
\text{SubMarkov( E, L, nil , E ) } ;
\]

\[
\text{Formula Procedure Prefix( X, L) ; Formula } X ; \text{List(any) } L ;
\]
\[
\text{Prefix } \leftarrow \text{cons('Pair(any)')(, X, L)} ;
\]
Procedure SubMarkov(E, L, S, O); Formula E, L, S, O;

begin  Formula K;  K ← Apply ( E, L );
comment:  E is the current subexpression under transformation,
          L is the list of transformations in the Markov Algorithm,
          S is a stack(list) of subexpressions to be processed next,
          O is a copy of the original expression, and
          signal is a global Boolean variable;

if signal then  begin  E ← K;  SubMarkov(O, L, S, O) ;  end else
if E == 'BinaryFormula then
    SubMarkov(LeftOperand(E), L, Prefix(RightOperand(E), S), O) else
if E == 'UnaryFormula then  SubMarkov(Operand(E), L, S, O) else
if ~Nullist(S) then  SubMarkov(Head(S), L, Tail(S), O) ;
end ;

Formula Procedure Apply(E, L); Formula E, L;

begin  Formula H ; H ← L ;
loop:  if Nullist(H) then go to fail else
    begin  if  Inst (E, LeftOperand(Head(H))) then
        begin  Apply ← Replace(RightOperand(Head(H))) ;
            signal ← true;  go to succeed ;  end else
            begin  H ← Tail (H) ;  go to loop  end ;
   fail :  signal ← false;  Apply ← E ;
succeed:  ;
end ;
Evaluation

The following procedure, and its associated subroutine, defines evaluation. Evaluation of a formula carries out arithmetic on constants, performs array accesses or procedure calls, applies lambda formulae to their operands, and carries out pattern matching.

**Formula Procedure** `eval(F)`; Formula F

```
if F == 'BinaryFormula then
    begin string O ; Formula L, R ; O ← Operator(F) ;
    L ← LeftOperand(F) ; R ← RightOperand(F) ;
    if O = '[' then eval ← ArrayAccess(eval(L), eval(R)) else
    if O = ',' then eval ← cons('Pair(any))(eval(L), eval(R)) else
    if O = '(' then
        begin
        if identifier(L) then eval ← Call(L, eval(R))
        if LambdaFormula(L) then
eval ← eval(subst(BoundVariableList(L), RightOperand(L), R))
        if ArrayFormula(L) eval ← Call(eval(L), eval(R)) ;
        end else
    if O = 'λ' then eval ← F else
    if O = '==' then eval ← Inst(eval(L), eval(R)) else
    if O = '=' then eval ← ExactlyEqual(eval(L), eval(R)) else
    if Atom(L) ∧ Atom(R) then eval ← PrimitiveEval(F) else eval ← F;
    end else
if F == 'UnaryFormula then
    begin if Operator(F) = '.' then eval ← Operand(F) else
if Operator(F) = '_' then eval ← F else
    eval ← PrimitiveEval(F) ;
    end
```
PrimitiveEval accepts a unary or binary formula whose operands are atomic and evaluates it to an atom if it can (by carrying out arithmetic or Boolean operations).

```
PrimitiveEval accepts a unary or binary formula whose operands are atomic and evaluates it to an atom if it can (by carrying out arithmetic or Boolean operations).

Formula Procedure PrimitiveEval (F) ; Formula F ;
begin Boolean BL, BR ; real RL, RR ; integer IL, IR ;
if F == "BinaryFormula" then
  begin Formula L, R ; string O ; O ← Operator(F) ;
    L ← LeftOperand(F) ; R ← RightOperand(F) ;
    if Boolean(L) ^ Boolean(R) then
      begin BL ← L ; BR ← R ;
        PrimitiveEval ← if O = '!' then BL v BR else
                      if O = '^' then BL ^ BR else
                      if O = '¬' then ~BL v BR else
                      Print('PrimitiveEval undefined') ;
                      go to exit ;
      end ;
    if (real(L) v integer(L)) ^ (real(R) v integer(R)) ^ ~ (integer(L) ^
      integer(R)) then begin
      if real(L) ^ real(R) then begin RL ← L ; RR ← R end else
      if real(L) ^ integer(R) then begin RL ← L ; IR ← R ; RR ← IR ; end else
      if integer(L) ^ real(R) then begin IL ← L ; RL ← IL ; RR ← R end
      PrimitiveEval ← if O = '+' then RL + RR else
                    if O = '-' then RL - RR else
                    if O = '×' then RL × RR else
                    if O = '/' then RL / RR else
                    if O = '↑' then RL ↑ RR else
                    if O = '>' then RL > RR else
                    if O = '<' then RL < RR else
                    if O = '=' then RL = RR else
                    Print('PrimitiveEval undefined') ;
                    go to exit ;
    end ;
```
if integer(L) \land integer(R) then
    begin IL \leftarrow L; IR \leftarrow R;
    PrimitiveEval \leftarrow \begin{cases} 
    \text{if } O = '+' \text{ then } IL + IR & \text{else} \\
    \text{if } O = '-' \text{ then } IL - IR & \text{else} \\
    \text{if } O = 'X' \text{ then } IL \times IR & \text{else} \\
    \text{if } O = '/' \text{ then } IL / IR & \text{else} \\
    \text{if } O = '^' \text{ then } IL \uparrow IR & \text{else} \\
    \text{if } O = '>' \text{ then } IL > IR & \text{else} \\
    \text{if } O = '<' \text{ then } IL < IR & \text{else} \\
    \text{if } O = '=' \text{ then } IL = IR & \text{else} \\
    \text{Print('PrimitiveEval undefined');} & \text{go to exit;}
    \end{cases}
    \end{end BinaryFormulae;}

    \begin{end if F == 'UnaryFormula then}
    \begin{begin Formula L \; \text{string } O ; \text{ } O \leftarrow \text{Operator(F); } L \leftarrow \text{Operand(F);}
    \begin{end if Boolean(L) then}
    \begin{begin BL \leftarrow L ;
    PrimitiveEval \leftarrow \begin{cases} 
    \text{if } O = '\sim' \text{ then } \sim BL & \text{else} \\
    \text{Print('PrimitiveEval undefined');} \; \text{go to exit2;}
    \end{end else}
    \end{end end Boolean(L) then}
    \begin{end if real(L) then}
    \begin{begin RL \leftarrow L ;
    PrimitiveEval \leftarrow \begin{cases} 
    \text{if } O = '+' \text{ then } RL & \text{else} \\
    \text{if } O = '-' \text{ then } -RL & \text{else} \\
    \text{if } O = 'sin' \text{ then } \sin(RL) & \text{else} \\
    \text{if } O = 'cos' \text{ then } \cos(RL) & \text{else}
    \end{end BinaryFormulae;}
    \end{end end if real(L) then}
if O = 'tan' then tan(RL) else
if O = 'ln' then ln(RL) else
if O = 'exp' then exp(RL) else
if O = 'arctan' then arctan(RL) else
Print('PrimitiveEval undefined');
go to exit2;

end;

if integer(L) then
begin IL ← L;
PrimitiveEval ← if O = '+' then IL else
if O = '-' then -IL else
if O = 'sin' then sin(IL) else
if O = 'cos' then cos(IL) else
if O = 'tan' then tan(IL) else
if O = 'ln' then ln(IL) else
if O = 'exp' then exp(IL) else
if O = 'arctan' then arctan(IL) else
Print('PrimitiveEval undefined');
go to exit2;
end;

PrimitiveEval ← F;
exit2: ;
end UnaryFormulae;
end PrimitiveEval;
2. List Processing

a. One Way Lists

In the following set of definitions an **Atom** is taken as undefined. It may be defined by supplying its descriptor. Lists are constructed recursively from atoms and lists as in Lisp [40].

**descriptor** Atom, Element;
**descriptor procedure** List;

\[
\text{List} \leftarrow \text{Nullist}::[\text{nil}] \lor \text{Pair}::[\text{Head}::\text{Element} \mid \text{Tail}::\text{List}]
\]

Element \leftarrow Atom \lor List;

As a result of these data structure definitions the following functions are defined automatically:

**Constructors**
- \text{cons} (Pair) \(x, y\)
- \text{cons} (Nullist)

**Selectors**
- \text{Head} (x)
- \text{Tail} (x)

**Predicates**
- \text{Atom} (x);
- \text{Nullist} (x)
- \text{Pair} (x)
- \text{List} (x)
- \text{x = y}

These functions may be employed, as in the following set of function definitions, to construct a fairly versatile set of operations over lists. Several of these functions are similar to those found in Lisp [40] and Formula Algol [49].
List Procedure Reverse(L) ; List L;
    begin List R ; R ← cons(Nullist);
    loop: if Nullist(L) then go to exit else
        R ← cons(Pair)(Head(L), R) ; L ← Tail(L); go to loop;
    exit : Reverse ← R ;
    end ;

List Procedure Append(f, g) ; List f, g;
    Append ← if Nullist(g) then cons(Pair)(f, cons(Nullist)) else
        cons(Pair)(Head(g), Append(f, Tail(g))) ;

Boolean Procedure Member(f, g); List f, g;
    Member ← if Nullist(g) then false else
        if f = Head(g) then true else Member(f, Tail(g)) ;

comment : The following procedure substitutes an instance of the atom f
for each instance of the atom g in the list structure h ;

List Procedure Subst(f, g, h) ; List f, g, h;
    if Atom(h) then Subst ← if h=g then f else h
    else Subst ← cons(Pair)(Subst(f, g, Head(h)), Subst(f, g, Tail(h))) ;

comment: The following procedure selects the n th object on the list L ;

List Procedure Nth(n, L) ; integer n ; List L ;
    Nth ← if n = 1 then Head(L) else Nth(n-1, Tail(L)) ;

List Procedure Concatenate(f, g) ; List f, g ;
    Concatenate ← if Nullist(f) then g else
        cons(Pair)(Head(f), Concatenate(Tail(f), g)) ;
comment: The following procedure alters the n th object in the list x to y;

List Procedure Alter(n, x, y) ; integer n ; List x, y ;

Alter ← if n = 1 then cons(Pair)(y, Tail(x))
else cons(Pair)(Head(x), Alter(n-1, Tail(x), y)) ;

comment: The following procedure gives a list of pairs formed from corresponding elements of the lists x and y and appends this list to the list a;

List Procedure Pairlis(x, y, a) ; List x, y, a ;

Pairlis ← if Nullist(x) then a else cons(Pair)(cons(Pair)(Head(x), Head(y)), Pairlis(Tail(x), Tail(y), a));

comment: Given that a is a list of pairs the following procedure will produce the first pair in a whose first member is x;

List Procedure Assoc(x, a) ; element x ; List a ;

Assoc ← if x = Head(Head(a)) then Head(a) else Assoc(x, Tail(a));

integer Procedure Length(L) ; List L ;

Length ← if Nullist(L) then 0 else 1 + Length(Tail(L)) ;

comment: The following procedure inserts x after the n th element of the list y;

List Procedure Insert(x, n, y) ; integer n ; element x ; List y ;

Insert ← if n = 0 then cons(Pair)(x, y) else cons(Pair)(Head(y), Insert(x, n-1, Tail(y)) ;
comment: The following procedure deletes the n th element of the list x;

List Procedure Delete(n, x); integer n; List x;
Delete ← if n = 1 then Tail(x) else
cons(Pair)(Head(x), Delete(n-1, Tail(x)));

comment : The following is the last procedure ;

List Procedure Last(x); List x;
Last ← if Nullist(Tail(x)) then x else Last(Tail(x)) ;

b. Symmetric Lists (cf. [41])

Because they permit the use of recursion, elementary descriptors are capable of describing, in finite closed form, tree structures of arbitrary depth. They are not capable of describing more complex kinds of well formedness such as loop connectedness and other varieties of graphical properties. These latter types of well formedness are capable of being described by predicate modifiers, however. For example, in the definition of a Symmetric List Element, given below, the elementary descriptor of the form

$$D ← [\text{datum: any} | \text{ll: .D} | \text{rl: .D}]$$

(where ll and rl stand for leftlink and rightlink respectively) describes only the well formedness of the format of a Symmetric List Element
but does not describe the manner in which such formats must be connected in a global graphical structure. For example, using

<table>
<thead>
<tr>
<th>datum</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
</tr>
<tr>
<td>rl</td>
</tr>
</tbody>
</table>

to represent the format given by D above

the following structure is well formed according to D:

![Diagram of well-formed structure]

Even though the individual elements are well formed in this diagram they are not connected to one another in the usual symmetric linkage associated with symmetric lists, in which each element points both to its right and left neighbors except for the end points which are nullists. In the definition of Symmetric List Element below it is seen that the predicate modifier \( \{ \lambda (y) \ (\text{Nullist}(ll(y)) \land \text{rl}(ll(y)) = y) \land (\text{Nullist}(rl(y)) \land ll(rl(y)) = y) \} \) supplies additional conditions for well formedness of the linkages given in each non null element.

The constructor modifier is also employed to provide a compact way of forming reverse symmetric linkages to elements previously constructed.

\[\text{descriptor SymmetricListElement ;} \]

\[\text{SymmetricListElement } \leftarrow \]

\[\text{Nullist} : [\text{nil}] \land \]

\[\text{Triple} : [\ \text{datum} : \text{any} \ | \ ll : .\text{SymmetricListElement} \ | \]

\[rl : .\text{SymmetricListElement} \ ]\]

\[\triangle \{ \lambda (x) \ \text{if} \ \sim \text{Nullist}(rl(x)) \ \text{then} \ ll(rl(x)) \leftarrow x \} \]
\[ \exists \{ \lambda (y) (\text{Nullist}(\text{ll}(y)) \vee \text{rl}(\text{ll}(y)) = y) \land (\text{Nullist}(\text{rl}(y)) \vee \text{ll}(\text{rl}(y)) = y) \} \]

SymmetricListElement Procedure C(L); List L;

\begin{align*}
C & \leftarrow \text{if} \ \text{Nullist}(L) \ \text{then} \ \text{cons}(\text{Nullist}) \ \text{else} \\
& \text{cons}(\text{Triple}) \ (\text{Head}(L) , \ \text{cons}(\text{Nullist}) , \ C(\text{Tail}(L))) ;
\end{align*}

comment: The following procedure deletes the \( n \)th element of the
symmetric list \( L \);

Procedure Delete(n, L); integer n; Triple L;

\begin{align*}
\text{if} \ n = 1 \ \text{then} \\
\text{begin} \ \text{if} \ \sim \text{Nullist}(\text{rl}(L)) \ \text{then} \ \text{ll}(\text{rl}(L)) \leftarrow \text{ll}(L) ; \\
\text{if} \ \sim \text{Nullist}(\text{ll}(L)) \ \text{then} \ \text{rl}(\text{ll}(L)) \leftarrow \text{rl}(L) ; \ \text{end} \\
\text{else} \ \text{Delete} \ (n-1, \ \text{rl}(L)) ;
\end{align*}

comment: The following procedure inserts \( x \) after the \( n \)th element of
the symmetric list \( L \);

Procedure Insert(x, n, L); integer n; Triple x, L;

\begin{align*}
\text{if} \ n = 1 \ \text{then} \\
\text{begin} \ \text{rl}(x) \leftarrow \text{rl}(L) ; \ \text{ll}(x) \leftarrow L ; \ \text{rl}(L) \leftarrow x ; \\
\text{if} \ \sim \text{Nullist}(\text{rl}(x)) \ \text{then} \ \text{ll}(\text{rl}(x)) \leftarrow x ; \\
\text{end} ; \\
\text{else} \ \text{Insert}(x, n-1, \ \text{rl}(L)) ;
\end{align*}

The procedure \( C(L) \) above maps a one way list \( L \) into a corresponding
symmetric list \( C(L) \). Let us trace through an example of its operation.
Let \( L \) be the list \((a, b, c)\) [constructed as a one way list with the construction \( \text{cons(Pair)}(a, \text{cons(Pair)}(b, \text{cons(Pair)}(c, \text{cons(Nullist)}))) \)] . Then \( C(L) = C((a, b, c)) = \text{cons(Triple)}(a, \text{cons(Nullist)}, C((b, c))) \) where \( C((b, c)) = \text{cons(Triple)}(b, \text{cons(Nullist)}, C((c))) \), and where \( C((c)) = \text{cons(Triple)}(c, \text{cons(Nullist)}, \text{cons(Nullist)}) \).

We see that the constructor for Triple applies the constructor modifier to each of the three triples constructed:

\[
\lambda (x) \text{ if } \sim \text{Nullist}(\text{rl}(x)) \text{ then } \text{ll}(\text{rl}(x)) \leftarrow x \tag{C((c))}
\]

has no effect on \( C((c)) \) since \( \text{rl}(C((c))) \) is a nullist,

\[
\lambda (x) \text{ if } \sim \text{Nullist}(\text{rl}(x)) \text{ then } \text{ll}(\text{rl}(x)) \leftarrow x \tag{C((b, c))}
\]

causes the left link of \( C((c)) \) to be linked symmetrically to \( C((b, c)) \), and

\[
\lambda (x) \text{ if } \sim \text{Nullist}(\text{rl}(x)) \text{ then } \text{ll}(\text{rl}(x)) \leftarrow x \tag{C((a, b, c))}
\]

causes the left link of \( C((b, c)) \) to be linked symmetrically to \( C((a, b, c)) \).

Thus we see that functional nesting links Triples to the right and the action of the constructor modifier links the nested Triples to the left. Linking is thus provided automatically in the definition of \( C(L) \) by utilization of the constructor modifier.

c. Threaded Lists (cf. [46])

Threaded lists, as given here, are composed of three sorts
of elements: Headers, which point to sublists; Data Elements, which refer to data and link to successor elements; and Terminators, which point back to Headers originating the chains which link to them. The data definitions are given as follows:

`descriptor Header, DataElement, Terminator, TLE;`

`Header ← top::[pointer: TLE] ← nontop::[pointer:TLE|link : TLE ];`

`DataElement ← [ datum : any |link : TLE ];`

`Terminator ← [ link: TLE ];`

`TLE ← Header ← DataElement ← Terminator;`

The procedure Maplist, given below, maps a one way list structure $L$ onto a corresponding threaded list. For example, using

<table>
<thead>
<tr>
<th>datum</th>
<th>link</th>
</tr>
</thead>
</table>

$\boxed{\text{pointer|link}}$ or $\boxed{\text{pointer}}$ to represent Header's, the list structure $(a, b, (c, d, e), f)$ would be mapped by Maplist onto the threaded list represented by

![Diagram of threaded list structure]
Maplist is defined as follows using the procedure M(L, T) as an auxiliary:

**TLE Procedure** Maplist(L); List(L);

begin TLE x, y;
x ← cons(top); y ← M(L, x); pointer(x) ← y;
Maplist ← x;
end;

**TLE Procedure** M(L, T); List L; Header T;

begin TLE x, y;
if Pair(L) then
  begin if Atom(Head(L)) then
    M ← cons(DataElement)(Head(L), M(Tail(L), T))
  else begin x ← cons(nontop)(, M(Tail(L), T));
    y ← M(Head(L), x); pointer(x) ← y; M ← x; end;
  end
else M ← cons(Terminator)(T);
end;

d. **Rings**

The procedure MakeRing below maps one way lists into rings with the aid of two auxiliary procedures Convert and Last. The basic data structure is the RingElement, abbreviated RE.
3. Mapping Algol Text into Flow Charts

In selecting a data structure to represent a subset of Algol

text we use the syntax of Algol as a point of departure. In reality,
as we soon show formally, the Backus Normal Form syntax of
Algol is a specification of a data structure for Algol text. It
specifies a data structure because it asserts the manner in which
any Algol string is to be decomposed and, at the same time, supplies
the units of the decomposition. For example, one decomposition
of the expression \( a \uparrow (b + c \times d) \) may be performed according
to the format \(<\text{factor}> \uparrow <\text{primary}>\) in which the unit \(<\text{factor}>\)
matches the subexpression \(a\), and in which the unit \(<\text{primary}>\) matches
the subexpression \( (b + c \times d) \). In general, any BNF syntax equation gives for each non-terminal unit of the language the alternate formats that define it, each format being a linear composition of two sorts of components: terminal characters (of type string) or other defined non-terminal units. The argument that BNF grammars define data structures can best be made, however, by exhibiting a mapping (amounting mostly to a change of punctuation) which shows that BNF can be modelled by a subset of the data definition facility. Specifically, let \( G \) be a BNF grammar and define a mapping \( M \) as follows: For every non-terminal \(<x>\) of \( G \), declare \( x \) a descriptor variable and set \( M(<x>) = x \). For every terminal character \( t \) of \( G \) set \( M(t) = ('t') \). For each string \( \theta = s_1s_2...s_n \) used in a BNF production set \( M(\theta) = [M(s_1) | M(s_2) | ... | M(s_n)] \) and for each production \(<x>::= \theta_1 | \theta_2 | ... | \theta_k\) with \( \theta_i \ (1 \leq i \leq k) \) strings, set \( M(<x>::= \theta_1 | \theta_2 | ... | \theta_k) = x \leftarrow M(\theta_1) \lor M(\theta_2) \lor ... \lor M(\theta_k) \). An example of the application of \( M \) is \( M(\ < \text{term} > ::= \ < \text{factor} > | \ < \text{term} > \times \ < \text{factor} > ) = \text{term} \leftarrow \text{[factor]} \lor \text{[term} \ | \ ('t' \times') \ | \text{factor} \] .

For the sake of simplicity in the task of mapping Algol text into flow charts, we would like to select a subset of the Algol syntax
and then compress it into a manageable data structure which retains somewhat the structure of Algol, yet requires, because of its reduced variability, less case analysis in the algorithm that maps Algol text into flow charts. To perform this compression so as not to violate Algol structure too severely, we agree to restrict ourselves to the use of certain operations. One of these is the class of string homomorphisms. A string homomorphism $H$ from Algol syntax into a reduced syntax is defined as follows: Given that $H$ is defined from terminals and non-terminals of Algol into terminals and non-terminals of a reduced syntax, then if $\theta = s_1 s_2 \ldots s_n$ is a string chosen from an Algol production, $H(\theta) = H(s_1) H(s_2) \ldots H(s_n)$ is a string in the compressed syntax (where $H(s_i)$ may be the empty string $\Lambda$ with the property that under concatenation $\Lambda x = x\Lambda = x$ for all strings $x$).

For example, we define a string homomorphism $H$ with the following properties: (a) if $t$ is a terminal character of Algol then $H(t) = \Lambda$, (b) if $x$ is $<\text{arithmetic expression}>$, $<\text{Boolean expression}>$, $<\text{subscript expression}>$, $<\text{function designator}>$, $<\text{variable}>$, $<\text{type}>$ or $<\text{left part list}>$ then $H(x) = \text{formula}$, and (c) if $x$ is $<\text{assignment statement}>$ then $H(x) = \text{assignment}$. Then, in giving the compressed syntax for an assignment statement, we
first take the syntax given in the Revised Algol Report, (cf. [43], p. 9, section 4.2.1):

\[
<\text{assignment statement}> ::= \langle\text{left part list}\rangle \langle\text{arithmetic expression}\rangle \\
| \langle\text{left part list}\rangle \langle\text{Boolean expression}\rangle.
\]

By applying \( H \) to this syntax we get:

\[
H(<\text{assignment statement}>) ::= H(<\text{left part list}>) \\
H(<\text{arithmetic expression}>) | H(<\text{left part list}>) \\
H(<\text{Boolean expression}>) .
\]

This further reduces to

\[
\text{assignment} ::= \text{formula} \text{formula} | \text{formula} \text{formula} .
\]

If we now map this homomorphically image into data structure notation by applying the mapping \( M \), defined above, we get

\[
\text{assignment} \leftarrow [\text{formula|formula}] \lor [\text{formula|formula}].
\]

The latter data structure description is redundant, and can be replaced by

\[
\text{assignment} \leftarrow [\text{lhs: formula | rhs : formula}]
\]

where the names \( \text{lhs} \) and \( \text{rhs} \) stand for left hand side and right hand side respectively. Another example is \( H(<\text{if clause}>) = \\
H(\text{if} <\text{Boolean expression} > \text{then}) = H(\text{if}) H(<\text{Boolean expression}>) H(\text{then}) \\
= \Lambda H(<\text{Boolean expression}>) \Lambda = H(<\text{Boolean expression}>) = \).

formula. This is used in the definition of the data structure
for a conditional (short for conditional statement) given below.

Another form of simplifying mapping we might use in devising
a data representation for a compressed Algol is one which provides
a standard representation for various sorts of lists in Algol, e.g.
identifier lists, subscript lists, for lists, etc. We first note that
the following two syntactic structures are ubiquitous in the syntax
of Algol: \(< \text{x-list} \.ud s := \text{x} | \text{x-list} \.ud x \) or \(< \text{x-list} \.ud s := \text{x} | \text{x} \ud x \pi \text{x-list} \)
where \(\pi\) is some punctuation mark (such as , or ; or :)
and where \(\text{x}\) is some unit such as <identifier>, <bound pair>,
<for list element>, <left part>, <subscript expression>,
<actual parameter> and so forth. Our data structure for Algol
is simplified if we agree to use the data structure List(\(\text{x}\)), where
\(\text{x}\) is an arbitrary type, to model all such lists. (cf. Section 2).

The application of string homomorphisms and the agreement
to use a uniform representation for lists in reducing a subset of
Algol leads to the following set of descriptors for a data structure
for Algol text:

descriptor conditional, goto, assignment, declaration, statement,
labelledstatement, block;

conditional ← [condition: formula | truearm: any | falsearm: any ];
goto ← [ label : identifier ];
assignment ← [ lhs: formula | rhs: formula ];
declaration ← [ declarator : formula | declarandlist: List(identifier)];
labelledstatement ← [ label : identifier | text : statement ];
block ← [ head : List(declaration) | body: List(statement) ] ;
statement ← conditional ∨ goto ∨ assignment ∨ labelledstatement ∨
   block ∨ dummy::[nil] ;

As is seen from these descriptors, the subset of Algol text does not
include procedures or procedure calls. The formula expressions in
assignments and conditionals are assumed to be represented by
constructions of the type given in Section 1 (on formula manipulation).

Flow charts [8, 33, 57] are constructed from three kinds of boxes:
action boxes, which contain executable simple statements (such as
assignments); decision boxes which perform a test and branch to a
true exit or a false exit, and null boxes which execute nothing but
form connections (these will be removed in the final flowchart produced
in the example below but serve as intermediary constructions).

The data structure for flow charts is as follows:

descriptor flowchart, actionbox, decisionbox ;
actionbox ← [ body : formula | exit : flowchart | mark: Boolean ] ;
decisionbox ← [ test : formula | trueexit: flowchart | falseexit: flowchart ] ;
flowchart ← actionbox ∨ decisionbox ∨ nullbox ::[nil] ;

The data structure for actionbox bears an additional component [mark:Boolean]
which is used in flowchart scanning procedures to prevent rescanning
previously scanned portions of a flowchart (setting the mark false prevents
redundant scanning). A pictorial representation of an actionbox is
given by a rectangular box with an arrow pointing to the exit or successor box in the flow chart. The pictorial representation of a decision box is given by a diamond box containing the test and the true and false exits are arrows marked Y (for yes) and N (for no):

The mapping of a representation of a piece of Algol text $S$ into a flow chart is performed by the function $\text{ConsFlowChart}(S)$ given below. $\text{ConsFlowChart}(S)$ uses standard label table techniques for handling undefined forward references. Namely, when a label is encountered it is placed in a table together with a signal that it has been encountered and the value of the translated structure it references. If a reference to a label not yet encountered occurs, then the label is entered in the table together with a signal that it has not been encountered and a chain to all undefined references. Later when the label is defined the chain is used to access the places where references must be filled in. The auxiliary procedure $\text{Join}(x, y)$ is used to connect two pieces of flowcharts $x$ and $y$.

The procedure $\text{ConsFlowChart}(S)$ and its associated auxiliary procedures are given on the following pages: All constructions in $S$ are assumed to be constructed as tagged data so that tests for
tags are adequate to separate the constructions encountered into cases.

**FlowChart**

**Procedure** ConsFlowChart(S) ;

value S ; statement S ;

begin flowchart x, y ; identifier L ;

if S == 'block' then

begin

x ← cons(actionbox) ; exit(x) ← 'next' ;
body(x) ← head(S) ; mark(x) ← true ;
y ← ConsFlowChart(body(S)) ;
ConsFlowChart ← Join(x, y) ;
go to exit ;
end ;

if S == 'assignment' then

begin

x ← cons(actionbox) ; exit(x) ← 'next' ;
body(x) ← S ; mark(x) ← true ;
ConsFlowChart ← x ;
go to exit ;
end ;

if S == 'goto' then

begin

x ← cons(actionbox) ; L ← label(S) ; exit(x) ← 'next' ;
body(x) ← nil ; mark(x) ← false ; Assign2(L, x) ;
ConsFlowChart ← x ;
go to exit ;
end ;

if S == 'conditional' then

begin

x ← cons(decisionbox) ; test(x) ← condition(S) ;
trueexit(x) ← ConsFlowChart(truearm(S)) ;
falseexit(x) ← ConsFlowChart(falsearm(S)) ;
ConsFlowChart ← x ;
go to exit ;
end ;

if S == 'labelledstatement' then

begin

L ← label(S) ; x ← ConsFlowChart(text(S)) ;
Assign1(L, x) ;
ConsFlowChart ← x ;
go to exit ;
end ;
if \((S == \text{nil}) \lor (S == \text{formula})\) then
begin
x ← cons(actionbox) ; exit(x) ← 'next' ;
body(x) ← S ; mark(x) ← true ;
go to exit ;
end ;

if \(S == \text{List} \text{(statement)}\) then
if \(S == \text{nil}\) then ConsFlowChart ← nil ;
else ConsFlowChart ←
Join(ConsFlowChart(head(S)), ConsFlowChart(tail(S))) ;
exit ;
end procedure ConsFlowChart ;

FlowChart Procedure Join( f₁ , f₂ ) ; flowchart f₁ , f₂ ;
comment : Briefly, Join replaces all occurrences of the string 'next'
in f₁ with references to f₂ unless f₁ or f₂ is a nullbox (i.e. is nil) in which case nil acts as an identity under the Join operation ;
begin
if \(f₁ == \text{nil}\) then begin Join ← f₂ ; go to exit end else
if \(f₂ == \text{nil}\) then begin Join ← f₁ ; go to exit end else
if ( f₁ == actionbox) \land (mark(f₁) = true ) then
begin
if exit(f₁) = 'next' then
exit(f₁) ← f₂; else exit(f₁) ← Join(exit(f₁), f₂) ;
end ;
if \(f₁ == \text{decisionbox}\) then
begin
trueexit(f₁) ← Join(trueexit(f₁), f₂) ;
falseexit(f₁) ← Join(falseexit(f₁), f₂) ;
end ;
Join ← f₁ ;
exit ;
end ;
end procedure Join ;

List Procedure Chain(F , C) ; flowchart F ; List C ;
begin List x ;
x ← cons(Pair)(F , C) ; Chain ← x ;
end ;
comment: Let \( \text{LT}[1: N, 1: 3] \) be a formula array serving as a label table. \( \text{LT}[i, 1] \) contains the identifier for the \( i \) th label encountered in the Algol text, \( \text{LT}[i, 2] \) contains either a chain or nil, and \( \text{LT}[i, 3] \) contains the Boolean formula \text{false} if the chain hasn't been assigned, and \text{true} if it has. The procedure \( \text{Position}(L) \) gets the position (an integer \( i \)) of the label \( L \) in the label table if it is there, otherwise it puts it there and assigns it a position;

Procedure \( \text{Assign1}(L, F) \); identifier \( L \); flowchart \( F \);
\begin{verbatim}
begin  integer \( i \);
  \( i \leftarrow \text{Position}(L) \);
comment: If label hasn't been assigned then assign chain in \( \text{LT}[i, 2] \);
  if \( \text{LT}[i, 3] = \text{false} \) then begin \( \text{Assign3}(\text{LT}[i, 2], F) \); \text{go to exit}; end;
comment: If label has been assigned then it is used twice;
  \text{Print('error, label used twice')};
comment: Fill in label table with value of encountered label;
  \( \text{LT}[i, 3] \leftarrow \text{true} \);
exit: \( \text{LT}[i, 2] \leftarrow F \); \( \text{LT}[i, 3] \leftarrow \text{true} \);
end
\end{verbatim}

comment: The following assignment algorithm is called when we meet a \text{go to}. There are two cases: (1) undefined forward references, and (2) references already defined;

Procedure \( \text{Assign2}(L, F) \); identifier \( L \); flowchart \( F \);
\begin{verbatim}
begin  \( i \leftarrow \text{Position}(L) \);
  if \( \text{LT}[i, 3] = \text{true} \) then \text{exit}(F) \leftarrow \text{LT}[i, 2] \) else
    \( \text{LT}[i, 2] \leftarrow \text{Chain}(F, \text{LT}[i, 2]) \);
end;
\end{verbatim}

Procedure \( \text{Assign3}(C, F) \); List \( C \); flowchart \( F \);
\begin{verbatim}
begin  List \( H \);
comment: Assign a chain;
loop: if \text{Nullist}(C) then \text{go to exit};
  \( H \leftarrow \text{Head}(C) \); \( C \leftarrow \text{Tail}(C) \); \text{exit}(H) \leftarrow F \);
  \text{go to loop};
exit: ;
end;
\end{verbatim}

integer Procedure \( \text{Position}(L) \); identifier \( L \);
\begin{verbatim}
begin  integer \( i \);
  for \( i \leftarrow 1 \) step 1 until \text{LTmax} do if \( \text{LT}[i, 1] = L \) then \text{go to exit}: \text{comment: If control reaches this point, label not yet in table;}
  \( \text{LT}[\text{LTmax}, 1] \leftarrow L \); \( \text{LT}[\text{LTmax}, 2] \leftarrow \text{nil} \); \( \text{LT}[\text{LTmax}, 3] \leftarrow \text{false} \);
  \text{LTmax} \leftarrow \text{LTmax} + 1;
  \text{exit: \text{Position} \leftarrow i} \);
end;
\end{verbatim}
Worked Example

The following short algorithm is a familiar algorithm used to put the maximum of an array \( A[1:N] \) of \( N \) numbers (\( N > 1 \)) in \( A[N] \). For brevity, assume that the value of \( i \) is initialized to zero on declaration. \( \text{Exch}(i, j) \) exchanges the values of \( A[i] \) and \( A[j] \).

\[
\begin{align*}
\text{begin} & \quad \text{integer } i ; \\
\text{loop} & \quad i \leftarrow i + 1 ; \\
& \text{if } i = N \text{ then go to exit ;} \\
& \text{if } A[i] > A[i+1] \text{ then Exch}[i, i+1] ; \\
& \text{go to loop ;} \\
\text{exit} & \quad ; \\
\text{end} ;
\end{align*}
\]

figure 1

To represent this algorithm as a data structure according to the definitions given for the data structure of Algol text above, the following sequence of constructions may be performed, where construction of formulae is assumed to be automatic for arithmetic expressions with dotted operators:

\[
\begin{align*}
\text{any} & \quad x, y, z, v, m, d, b ; \\
x & \leftarrow \text{cons('labelledstatement)} ; \text{label(x) } \rightarrow \text{.exit ; text(x) } \rightarrow \text{nil ;} \\
y & \leftarrow \text{cons(Pair)( x, cons(Nullist)) ;} \\
x & \leftarrow \text{cons('goto) ; label(x) } \rightarrow \text{.loop ;} \\
z & \leftarrow \text{cons(Pair)(x, y) ;}
\end{align*}
\]
\[ x \leftarrow \text{cons}'\text{conditional}'; \text{condition}(x) \leftarrow A.[i] > A.[i+1]; \]
\[ \text{truearm}(x) \leftarrow \text{Exch.}(i, i+1); \text{falsearm}(x) \leftarrow \text{nil}; \]
\[ y \leftarrow \text{cons}(\text{Pair})(x, z); \]
\[ x \leftarrow \text{cons}'\text{conditional}'; \text{condition}(x) \leftarrow (i = \text{N}); \text{falsearm}(x) \leftarrow \text{nil}; \]
\[ \text{truearm}(x) \leftarrow \text{cons}'\text{goto}'.\text{exit}'; \]
\[ z \leftarrow \text{cons}(\text{Pair})(x, y); \]
\[ x \leftarrow \text{cons}'\text{labelledstatement}'; \text{label}(x) \leftarrow \text{.loop}; \]
\[ v \leftarrow \text{cons}'\text{assignment}'; \text{lhs}(v) \leftarrow i; \text{rhs}(v) \leftarrow i + 1; \text{text}(x) \leftarrow v; \]
\[ y \leftarrow \text{cons}(\text{Pair})(x, z); \]
\[ m \leftarrow \text{cons}(\text{Pair})(i, \text{nil}); \]
\[ d \leftarrow \text{cons}'\text{declaration}'; \text{declarator}(d) \leftarrow \text{.integer}; \text{declarandlist}(d) \leftarrow m; \]
\[ m \leftarrow \text{cons}(\text{Pair})(d, \text{nil}); \]
\[ b \leftarrow \text{cons}'\text{block}'; \text{head}(b) \leftarrow m; \text{body}(b) \leftarrow y; \]

**figure 2**

The final data structure resulting from this sequence of constructions is the value of \( b \). The corresponding flowchart is obtained by evaluating \( \text{ConsFlowChart}(b) \). The data structure constructed by the sequence of constructions in figure 2 is represented pictorially in figure 3. The flowchart that results from evaluating \( \text{ConsFlowChart}(b) \) is represented pictorially in figure 4. Figure 5 gives a neater version of the flowchart pictured in figure 4, which flowchart is obtained by applying the procedure \( \text{Purify}(F) \) to the flowchart of figure 4. This procedure is defined below. Figure 6 gives a diagram of intermediate stages in the computation of the flowchart produced by evaluating \( \text{ConsFlowChart}(b) \).
### BLOCK

<table>
<thead>
<tr>
<th>HEAD</th>
<th>LIST (DECLARATION)</th>
<th>TAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECLARATOR</td>
<td>DECLARAND</td>
<td>LIST</td>
</tr>
<tr>
<td>LIST (IDENTIFIER)</td>
<td>HEAD</td>
<td>TAIL</td>
</tr>
<tr>
<td>begin</td>
<td>integer</td>
<td>i; nil</td>
</tr>
</tbody>
</table>

### LIST (STATEMENT)

<table>
<thead>
<tr>
<th>HEAD</th>
<th>LABEL</th>
<th>STATEMENT</th>
<th>TAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>LABEL</td>
<td>TEXT</td>
<td>TYPE</td>
<td>RHS</td>
</tr>
<tr>
<td>loop</td>
<td>i ←</td>
<td>i + 1</td>
<td></td>
</tr>
</tbody>
</table>

#### α

<table>
<thead>
<tr>
<th>HEAD</th>
<th>CONDITIONAL</th>
<th>TAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE ARM</td>
<td>FALSE ARM</td>
<td></td>
</tr>
<tr>
<td>if i = N then go to exit</td>
<td>nil</td>
<td></td>
</tr>
</tbody>
</table>

#### β

<table>
<thead>
<tr>
<th>HEAD</th>
<th>CONDITIONAL</th>
<th>TAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE ARM</td>
<td>FALSE ARM</td>
<td></td>
</tr>
<tr>
<td>if A[i] &gt; A[i+1] then Exch(i, i+1);</td>
<td>nil</td>
<td></td>
</tr>
</tbody>
</table>

#### γ

<table>
<thead>
<tr>
<th>HEAD</th>
<th>GO TO</th>
<th>TAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>LABEL</td>
<td>loop</td>
<td></td>
</tr>
</tbody>
</table>

#### δ

<table>
<thead>
<tr>
<th>HEAD</th>
<th>LABELLED</th>
<th>STATEMENT</th>
<th>TAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>LABEL</td>
<td>TEXT</td>
<td>exit</td>
<td>nil</td>
</tr>
</tbody>
</table>

---

*figure 3*
integer i

i ← i + 1

i = N ?

Y: nil

N: nil


Y: Exch(i, i+1)

N: nil

'next'
figure 5, Purified Flow Chart
AB - actionbox, DB - decisionbox, LT - label table

figure 6
The following procedure, while not perfectly general, removes all nil action boxes from the flow chart in figure 4 producing the flow chart in figure 5.

**FlowChart Procedure** Purify(F); flowchart F;

begin
  if ( F == "actionbox" ∧ (mark(F) = true) ) then
    begin
      if body(F) = nil then Purify ← Purify(exit(F))
      else begin
        exit(F) ← Purify(exit(F)) ;
        Purify ← F ;
        end ;
      go to exit ;
    end ;
  if ( F == "actionbox") ∧ (mark(F) = false ) then
    begin
      if body(F) = nil then Purify ← exit(F) else Purify ← F ;
      go to exit ;
    end ;
  if F == "decisionbox then
    begin
      trueexit(F) ← Purify(trueexit(F)) ;
      falseexit(F) ← Purify(falseexit(F)) ;
      end ;
  else Purify ← F ;
  exit ;
end procedure ;
4. Electronic Circuits

In the following example we give a procedure which computes a transfer function of a simple species of electrical networks synthesized from parallel and series combinations of resistors, capacitors and inductors. The particular transfer function chosen is the complex Laplace transform of the circuit impedance \( Z(s) = \frac{V(s)}{I(s)} \) (see references [59] and [65]).

An example of the use of the transfer function \( Z(s) \) is as follows. Let \( C \) be a circuit for which the complex Laplace transform of the impedance is \( Z_C(s) = \frac{V_C(s)}{I_C(s)} \). Suppose we know the voltage \( V(t) \) applied across \( C \) as a function of time \( t \).

Let \( V(s) = \mathcal{L}(V(t)) \) be the complex Laplace transform of \( V(t) \).

Then \( V(s) \times \frac{1}{Z_C(s)} = V(s) \times \frac{I_C(s)}{V_C(s)} = I(s) \), where \( I(s) \) is the complex Laplace transform of \( I(t) \) the current across \( C \) as a function of time \( t \). Upon taking the inverse Laplace transform of \( I(s) \) we get \( I(t) = \mathcal{L}^{-1}(I(s)) \).

There are simple rules for calculating \( Z(s) \) from simple networks synthesized from resistors, capacitors and inductors. Specifically, we can build up a mapping \( T(C) \) of a circuit \( C \) into a transfer function as follows: (1) If \( R \) is a resistance \( \frac{\text{\underline{---}}}{} \) then \( T(\frac{\text{\underline{---}}}{} \rightarrow R \), (2) If \( C \) is a capacitance
If \( C \) is an inductance, then 
\[
T \left( \frac{1}{s \times C} \right) 
\]

(3) If \( L \) is an inductance, then 
\[
T \left( \frac{1}{L \times s} \right) 
\]

(4) If \( a \) and \( b \) are arbitrary subcircuits combined in parallel, 
\[
T \left( \frac{T(a) \times T(b)}{T(a) + T(b)} \right) 
\]

(5) If \( a \) and \( b \) are arbitrary subcircuits combined in series, 
\[
T \left( \frac{T(a) \times T(b)}{T(a) + T(b)} \right) 
\]

Note: Several subcircuits combined in parallel may be represented as combinations of binary parallel circuits, e.g.

Note also that circuit combinations such as
may be represented equivalently by networks synthesized from binary series and parallel elements such as

However, no claim is made with respect to the power of combinations of binary series and parallel circuits to represent arbitrary networks. In fact, the author believes, but cannot prove, that they are not general. For instance, no equivalent circuit has been found for the circuit.

We may now define the data structures for the simple species of circuit synthesized from binary series and parallel combinations of resistors, capacitors and inductors as follows:

descriptor Part, Circuit;


Circuit ← Part ∨ SeriesElement ::[Circuit | Circuit] ∨ ParallelElement ::[Circuit | Circuit];

Given these data structure definitions the circuit in figure 7 can be represented by the following sequence of declarations and constructions:
The following procedure defines a function which obtains the impedance transfer function from a representation of a circuit $C$. 

```lisp
Circuit C; Formula r1, r2, r3, c1, c2, l1, l2; Part R1, R2, R3, C1, C2, L1, L2;
R1 ← cons(''Resistor)(r1);
R2 ← cons(''Resistor)(r2);
R3 ← cons(''Resistor)(r3);
C1 ← cons(''Capacitor)(c1);
C2 ← cons(''Capacitor)(c2);
L1 ← cons(''Inductor)(l1);
L2 ← cons(''Inductor)(l2);
C ← cons(''SeriesElement)(cons(''ParallelElement)
                             (cons(''SeriesElement)(cons(''ParallelElement)(R1, C1), L1),
                              cons(''SeriesElement)(R2, cons(''ParallelElement)(C2, L2)), R3);
```

**figure 7**

The following procedure defines a function which obtains the impedance transfer function from a representation of a circuit $C$. 

Formula Procedure \( T(C) \); Circuit \( C \);

\[
\begin{align*}
&\text{begin} \\
&\quad \text{if } C = "\text{ParallelElement} \text{ then } T \leftarrow \frac{T(C[1]) \times T(C[2])}{T(C[1]) + T(C[2])}; \quad \\
&\quad \text{if } C = "\text{SeriesElement} \text{ then } T \leftarrow T(C[1]) + T(C[2]); \quad \\
&\quad \text{if } C = "\text{Resistor} \text{ then } T \leftarrow C[1]; \quad \\
&\quad \text{if } C = "\text{Capacitor} \text{ then } T \leftarrow \frac{1}{C[1] \times s}; \quad \\
&\quad \text{if } C = "\text{Inductor} \text{ then } T \leftarrow C[1] \times s; \quad \\
&\text{end;} \\
\end{align*}
\]

In the body of this procedure dotted operators have been used as a contracted notation for construction of binary formulae. That is, if \( \theta \) represents a binary operator then \( A \cdot \theta B \) represents \( \text{cons(BinaryFormula)}('\theta', A, B) \).

If we use the abbreviation \( A \uparrow B \) to represent \( \text{cons('ParallelElement')(A, B)} \) and the abbreviation \( A-B \) to represent \( \text{cons('SeriesElement')(A, B)} \) then the circuit, in figure 7 above, can be represented by the formula \( C = ( (r1\uparrow c1) -11) \uparrow(r2 - (c2\uparrow 12)) - r3 ) \). This permits us to express the result of applying \( T \) to the circuit \( C \) to produce the transfer function \( T(C) \).

The sequence of reductions resulting from this application is as follows:

\[
T(C) = T\left((\((r1\uparrow c1)\uparrow-11)\uparrow(r2-(c2\uparrow 12))\)-r3\right) \\
= T\left((r1\uparrow c1 -11)\uparrow(r2- c2\uparrow 12)\right) + T(r3) \\
= \frac{T(r1\uparrow c1-11) \times T(r2-c2\uparrow 12)}{T(r1\uparrow c1-11) + T(r2-c2\uparrow 12)} + r3 \\
= \frac{[T(r1\uparrow c1) + T(l1)] \times [T(r2) + T(c2\uparrow 12)]}{[T(r1\uparrow c1) + T(l1)] + [T(r2) + T(c2\uparrow 12)]} + r3
\]
\[
\begin{align*}
\frac{T(r_1) \times T(c_1)}{T(r_1) + T(c_1)} + 11s & \left[ r_2 + \frac{T(c_2) \times T(12)}{T(c_2) + T(12)} \right] + r_3 \\
\frac{T(r_1) \times T(c_1)}{T(r_1) + T(c_1)} + 11s + r_2 + \frac{T(c_2) \times T(12)}{T(c_2) + T(12)} & \\
\frac{r_1 \times \frac{1}{(c_1s)}}{r_1 + \frac{1}{(c_1s)}} + 11s & \left[ r_2 + \frac{(1/(c_2s) \times 12s)}{1/(c_2s) + 12s} \right] + r_3 \\
\frac{r_1 \times \frac{1}{(c_1s)}}{r_1 + \frac{1}{(c_1s)}} + 11s + r_2 + \frac{(1/(c_2s) \times 12s)}{1/(c_2s) + 12s} &
\end{align*}
\]

A formula simplification routine [17] could then be applied to clear this fraction and reduce common factors in the result. The inverse Laplace transform might then be taken.

In this example, and elsewhere in this dissertation, we have introduced and used abbreviations in order to raise the issue of notation and to permit comparisons. It is our belief that once we have variable data structures we must also have variable syntax. This view is explored further in Chapter VII.

Given several languages with data structures adequate for a particular task, we should choose that language with the most convenient notation. The Formula Algol program on the next page performs T(C) and we see that it is capable of providing convenient notations for the circuits of this example. The Formula Algol representation of these circuits becomes strained, however, once we introduce circuits having loops which cannot be represented by binary trees. The flow chart example of the last section is a data space that strains the representational powers of formulae. If the data structures of a language are inadequate for a task, its notation buys us nothing. Yet once we are given well adapted data, we need well adapted notation.
BEGIN EXECUTION 13:22:43; 06641 AVAILABLE CELLS
CIRCUIT = (R1*C1 - L1)*R2 - C2*L2) - R3
TRANSFER - FUNCTION = (R1*1/(C1*S)/(R1 + 1/(C1*S)) + L1*S)
*
R2 + 1/(C2*S)*L2*S/(1/(C2*S) + L2*S))/(R1*1/(C1*S)/(R1 + 1/(C1*S)) + L1*S + R2 + 1/(C2*S)*L2*S/(1/(C2*S) + L2*S))
+ R3

TIME USED: 00:00:33 PAGES: 4
11562 13:22:46
5. Complex Variables

The following set of data definitions and procedures define complex numbers, complex multiplication and complex addition. A comparison of the procedure for complex multiplication below with McCarthy's example, given on page 19, reveals that both are essentially the same. This is probably because both examples say the minimum necessary to specify the task under the assumption that one starts with some sort of structure definition and that one works with the associated predicates, selectors and constructors.

descriptor complex;

complex ← [ realpart: real | imagpart: real ];

complex procedure cmult(u, v); complex u, v;
    cmult ← cons(complex)( realpart(u) × realpart(v) -
    imagpart(u)× imagpart(v),
    realpart(u) × imagpart(v) +
    imagpart(u)× realpart(v) );

complex procedure cadd(u, v); complex u, v;
    cadd ← cons(complex)( realpart(u) + realpart(v),
    imagpart(u)+ imagpart(v) );

Complex subtraction can be expressed as cadd(u, -v) where -v for v complex is performed by executing the parallel assignment realpart(v), imagpart(v) ← -realpart(v), -imagpart(v). Complex division can be expressed as cmult(u, conj(v))/cmult(v, conj(v)) where
conj(v) is a procedure which takes the conjugate of v defined as follows:

```lisp
complex procedure conj(v); complex v;
    conj ← cons(complex)(realpart(v), -imagpart(v));
```

6. Files

The following example exhibits a procedure for computing a boiling point index for a file of organic compounds. If an organic substance is a pure compound and its boiling point is measured, the boiling point can serve as an important clue to the identity of the compound since there are relatively few compounds with a given boiling point. Figures 8 and 9 show selections from a table of physical constants of organic compounds and from a boiling point index for this table respectively. These selections are taken from the Handbook of Chemistry and Physics [25] which lists (in its 37th edition) 8988 common organic compounds together with several of their physical constants and which provides a boiling point index for these compounds. Our data structure definitions will describe the table of physical constants of these compounds and a procedure, given below, is defined to act on this data structure to produce a boiling point index.
### PHYSICAL CONSTANTS OF

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Synonyms</th>
<th>Formula</th>
<th>Mol. Wt.</th>
</tr>
</thead>
<tbody>
<tr>
<td>560</td>
<td>Anesthesia</td>
<td>See Benzene.</td>
<td>C₇H₅N·H₂CO·C₆H₅</td>
<td>202.33</td>
</tr>
<tr>
<td>561</td>
<td>Analgin (1)</td>
<td>Schenckan, benzoylacetanilide, benzoylacetanilide.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>562</td>
<td>Analgin (2)</td>
<td>Formerly, Analgin.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>563</td>
<td>Anhydrocoline</td>
<td>See Analgin.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>564</td>
<td>Anhydroligneine</td>
<td>See Analga.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>565</td>
<td>Angelic acid</td>
<td>cis-3-Methyl-butenolic acid; hydroxytetrahydro-3-methyl-butenolic acid</td>
<td>C₇H₁₀O₂</td>
<td>146.19</td>
</tr>
<tr>
<td>566</td>
<td>Anhalonitidine</td>
<td></td>
<td>C₇H₆N₂O₃</td>
<td>223.27</td>
</tr>
<tr>
<td>567</td>
<td>Anhalonitidine hydrochloride</td>
<td></td>
<td>C₇H₆N₂O₃.HCl</td>
<td>267.71</td>
</tr>
<tr>
<td>568</td>
<td>dl-Anhydrocetonglucine</td>
<td></td>
<td>C₇H₆N₂O₃</td>
<td>167.20</td>
</tr>
<tr>
<td>569</td>
<td>dl-Anhydrocetonglucine hydrochloride</td>
<td></td>
<td>C₇H₆N₂O₃.HCl</td>
<td>203.57</td>
</tr>
<tr>
<td>570</td>
<td>Anhydroformaldehyde</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>571</td>
<td>Aniline</td>
<td>See o-Triazole, hexamethylenemine, anilinamine; (\text{C₆H₄(NH₂)}₂)</td>
<td>C₆H₁₂N₂</td>
<td>106.12</td>
</tr>
<tr>
<td>572</td>
<td>Anin</td>
<td>hydrochloride</td>
<td>C₆H₁₂N₂.HCl</td>
<td>120.50</td>
</tr>
<tr>
<td>573</td>
<td>Antipyrine</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>574</td>
<td>N-acetyl-</td>
<td>See Antipyrine, acetyl-, (\text{C₆H₄(NH₂)}₂)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>575</td>
<td>N-allyl-</td>
<td>See Antipyrine, allyl-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>576</td>
<td>N-propargylamine</td>
<td>See Antipyrine, propargylamine, (\text{C₆H₄(NH₂)}₂)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>577</td>
<td>N-propargylamine</td>
<td>See Antipyrine, propargylamine, (\text{C₆H₄(NH₂)}₂)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>578</td>
<td>N-propargylamine</td>
<td>See Antipyrine, propargylamine, (\text{C₆H₄(NH₂)}₂)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>579</td>
<td>N-propargylamine</td>
<td>See Antipyrine, propargylamine, (\text{C₆H₄(NH₂)}₂)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>580</td>
<td>N-propargylamine</td>
<td>See Antipyrine, propargylamine, (\text{C₆H₄(NH₂)}₂)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>581</td>
<td>N-propargylamine</td>
<td>See Antipyrine, propargylamine, (\text{C₆H₄(NH₂)}₂)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>582</td>
<td>N-propargylamine</td>
<td>See Antipyrine, propargylamine, (\text{C₆H₄(NH₂)}₂)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>583</td>
<td>N-propargylamine</td>
<td>See Antipyrine, propargylamine, (\text{C₆H₄(NH₂)}₂)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>584</td>
<td>N-propargylamine</td>
<td>See Antipyrine, propargylamine, (\text{C₆H₄(NH₂)}₂)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>585</td>
<td>N-propargylamine</td>
<td>See Antipyrine, propargylamine, (\text{C₆H₄(NH₂)}₂)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>586</td>
<td>N-propargylamine</td>
<td>See Antipyrine, propargylamine, (\text{C₆H₄(NH₂)}₂)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>587</td>
<td>N-propargylamine</td>
<td>See Antipyrine, propargylamine, (\text{C₆H₄(NH₂)}₂)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>588</td>
<td>N-propargylamine</td>
<td>See Antipyrine, propargylamine, (\text{C₆H₄(NH₂)}₂)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>589</td>
<td>N-propargylamine</td>
<td>See Antipyrine, propargylamine, (\text{C₆H₄(NH₂)}₂)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>590</td>
<td>N-propargylamine</td>
<td>See Antipyrine, propargylamine, (\text{C₆H₄(NH₂)}₂)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>591</td>
<td>N-propargylamine</td>
<td>See Antipyrine, propargylamine, (\text{C₆H₄(NH₂)}₂)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Name approved by the International Union of Chemistry.

---

### ORGANIC COMPOUNDS (Continued)

<table>
<thead>
<tr>
<th>No.</th>
<th>Crystalline form, color and index of refraction</th>
<th>Density g/ml</th>
<th>Melting point, °C</th>
<th>Boiling point, °C</th>
<th>Solubility in grams per 100 ml of Water</th>
<th>Alcohol</th>
<th>Ether, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>560</td>
<td>col.-yel. needle</td>
<td>210</td>
<td></td>
<td></td>
<td></td>
<td>v. s. a.</td>
<td>v. s. eth.</td>
</tr>
<tr>
<td>561</td>
<td>col. leaf f. f.</td>
<td>0.99344</td>
<td>22.5, fra. 235.3</td>
<td>v. s. a. 20-56%</td>
<td></td>
<td>s. eth.</td>
<td>h. acet.</td>
</tr>
<tr>
<td>562</td>
<td>col. monon. pr.</td>
<td>liq.</td>
<td></td>
<td></td>
<td></td>
<td>v. s. a.</td>
<td>v. s. eth.</td>
</tr>
<tr>
<td>563</td>
<td>col. monon. pr.</td>
<td>liq.</td>
<td></td>
<td></td>
<td></td>
<td>v. s. a.</td>
<td>v. s. eth.</td>
</tr>
<tr>
<td>564</td>
<td>col. monon. pr.</td>
<td>liq.</td>
<td>45</td>
<td>185</td>
<td></td>
<td>v. s. a.</td>
<td>v. s. eth.</td>
</tr>
<tr>
<td>565</td>
<td>col. monon. pr.</td>
<td>liq.</td>
<td>160</td>
<td></td>
<td></td>
<td>v. s. a.</td>
<td>v. s. eth.</td>
</tr>
<tr>
<td>566</td>
<td>col. monon. pr.</td>
<td>liq.</td>
<td>85</td>
<td></td>
<td></td>
<td>v. s. a.</td>
<td>v. s. eth.</td>
</tr>
<tr>
<td>567</td>
<td>wh. cr. powd.</td>
<td>&gt;230 t.</td>
<td></td>
<td></td>
<td></td>
<td>v. s. a.</td>
<td>v. s. a.</td>
</tr>
<tr>
<td>568</td>
<td>wh. cr. f. w</td>
<td>220-30 d.</td>
<td>0.235 d.</td>
<td></td>
<td></td>
<td>v. s. a.</td>
<td>v. s. eth.</td>
</tr>
<tr>
<td>569</td>
<td>rhomb. need f. al.</td>
<td>240-1</td>
<td></td>
<td></td>
<td></td>
<td>v. s. a.</td>
<td>v. s. eth.</td>
</tr>
<tr>
<td>570</td>
<td>rhomb. need f. al.</td>
<td>240-1</td>
<td></td>
<td></td>
<td></td>
<td>v. s. a.</td>
<td>v. s. eth.</td>
</tr>
<tr>
<td>571</td>
<td>rhomb. need f. al.</td>
<td>240-1</td>
<td></td>
<td></td>
<td></td>
<td>v. s. a.</td>
<td>v. s. eth.</td>
</tr>
<tr>
<td>572</td>
<td>rhomb. need f. al.</td>
<td>240-1</td>
<td></td>
<td></td>
<td></td>
<td>v. s. a.</td>
<td>v. s. eth.</td>
</tr>
<tr>
<td>573</td>
<td>leaf f. al.</td>
<td>1.02505</td>
<td>3.0%</td>
<td></td>
<td></td>
<td>v. s. a.</td>
<td>v. s. eth.</td>
</tr>
<tr>
<td>574</td>
<td>leaf f. al.</td>
<td>1.02505</td>
<td>3.0%</td>
<td></td>
<td></td>
<td>v. s. a.</td>
<td>v. s. eth.</td>
</tr>
<tr>
<td>575</td>
<td>leaf f. al.</td>
<td>1.02505</td>
<td>3.0%</td>
<td></td>
<td></td>
<td>v. s. a.</td>
<td>v. s. eth.</td>
</tr>
<tr>
<td>576</td>
<td>leaf f. al.</td>
<td>1.02505</td>
<td>3.0%</td>
<td></td>
<td></td>
<td>v. s. a.</td>
<td>v. s. eth.</td>
</tr>
<tr>
<td>577</td>
<td>leaf f. al.</td>
<td>1.02505</td>
<td>3.0%</td>
<td></td>
<td></td>
<td>v. s. a.</td>
<td>v. s. eth.</td>
</tr>
<tr>
<td>578</td>
<td>leaf f. al.</td>
<td>1.02505</td>
<td>3.0%</td>
<td></td>
<td></td>
<td>v. s. a.</td>
<td>v. s. eth.</td>
</tr>
<tr>
<td>579</td>
<td>leaf f. al.</td>
<td>1.02505</td>
<td>3.0%</td>
<td></td>
<td></td>
<td>v. s. a.</td>
<td>v. s. eth.</td>
</tr>
<tr>
<td>580</td>
<td>leaf f. al.</td>
<td>1.02505</td>
<td>3.0%</td>
<td></td>
<td></td>
<td>v. s. a.</td>
<td>v. s. eth.</td>
</tr>
<tr>
<td>581</td>
<td>leaf f. al.</td>
<td>1.02505</td>
<td>3.0%</td>
<td></td>
<td></td>
<td>v. s. a.</td>
<td>v. s. eth.</td>
</tr>
</tbody>
</table>

For explanation and abbreviations see beginning of table.

729
Data Structure of Organic Compound File

descriptor Record, OrganicCompoundFile;

Record ← [ Number: integer | Name: string | Synonyms: string |
Formula: string | MolWt: real | Properties: string |
Density: formula | MeltingPt: formula | BoilingPt: integer |
Solubility: [ inWater: formula | inAlcohol: formula | inOther: string ];

OrganicCompoundFile ← 8988 × [Record];

Notice, in figure 8 that each compound has a number, and that, in
figure 9, boiling points correspond to lists of compound numbers having
that boiling point. We will build up the boiling point index as a one
dimensional array of lists of compound numbers. Since boiling points
range from -190°C to 520°C, we will need an array of 710 entries,
one for each centigrade degree between -190 and 520. The data
structure for this array is as follows:

descriptor procedure Array(n,t); integer n: descriptor t;

Array ← n × [ t ];

procedure ComputeBoilingPointIndex
begin Array(710, List(integer)) A; integer i; Record R;
for i ← 1 step 1 until 8988 do
begin
R ← OrganicCompoundFile[i];
Append( A[BoilingPt(R) + 191], Number(R) );
end;
end ;
One can sequence through the array A and print non-null lists of compound numbers opposite their boiling points. This might be done as follows:

```plaintext
for i ← 1 step 1 until 710 do
    if ~Nulllist(A[i]) then Print( i-191, A[i] ) ;
```

Examples of data structures of other types of files are as follows:

**Medical Files**

```plaintext
descriptor MedicalFile, medicinespecifier, dietcode ;
    MedicalFile ← [patientname: string | roomnumber: integer |
    doctor: string | admissiondate: List(integer) |
    medicationschedule: List(medicinespecifier) | diet: dietcode ] ;

    medicinespecifier ← [medicinename: identifier | dosage: formula |
    schedule: List(string) ] ;

    dietcode ← {softsolid, liquid, intravenous, normal} ;
```

**Library Cards**

```plaintext
descriptor AuthorCard, SubjectCard, TitleCard, Book ;
    Book ← [Title: string | Author: string | Publisher: string | pages: integer |
    size: formula | date: formula | precis: List(string) ] ;

    AuthorCard ← [ author: string | referent: Book ] ;

    SubjectCard ← [ subject: string | referent: Book ] ;

    TitleCard ← [ title: string | referent: Book ] ;
```

Note that a card catalogue programmed using these data structure definitions need store only one common shared copy of the information about each book.
Abstract Files

descriptor procedure Field(n) ; integer n ; Field ← n × [ bit ] ;
descriptor procedure CharSeq(n) ; integer n ; CharSeq ← n × [character] ;
descriptor letter, digit, bit, character ;
bit ← { 0, 1 }; digit ← { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }; letter ← { a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z }; character ← letter ∨ digit ;
descriptor procedure UnsignedInt(n) ; integer n ; UnsignedInt ← n × [ digit ] ;
descriptor procedure SubRecord(n) ; integer n ;
   SubRecord ← Field(n) ∨ CharSeq(n) ∨ UnsignedInt(n) ;
descriptor procedure Record(m, n) ; integer m, n ;
   Record ← m × [ SubRecord(n) ] ;
descriptor procedure File (k, m, n) ; integer k, m, n ;
   File ← k × [ Record(m, n) ] ;

Note, if the sizes of files, records, and subrecords is not known
in advance of allocate time, then indefinite may be substituted in
place of k, m and n in the above definitions to provide this flexibility.
For example, File ← indefinite × [ Record ] ; Record ← indefinite ×[SubRecord]
and SubRecord ← indefinite × [ bit ∨ character ∨ digit ] ;

7. Matrices

An m by n rectangular matrix of data of type t may be
described by the use of replicators as follows:

descriptor procedure Matrix(m, n, t) ; integer m, n ; descriptor t ;
   Matrix ← m × [ n × [ t ] ] ;
Suppose that objects of type $t$ belong to a commutative ring $R$. Let $\text{Add}(t, x, y)$ be a general addition procedure which is a function of $t$ specifying how objects of type $t$ are to be added in $R$, let $\text{Mult}(t, x, y)$ be a general multiplication procedure specifying how objects of type $t$ are to be multiplied in $R$, and let $\text{zero}(t)$ give a zero or additive identity element for the additive subgroup of $R$. These procedures would give different results, for example, for $t$ of type \textbf{real}, \textbf{Boolean}, \textbf{complex} and \textbf{path} respectively. Then, as Berge points out (cf. [4] pages 138-140), matrix addition and matrix multiplication may be defined for matrices with elements in $R$, and may thus be defined independently of the particular type $t$ of their elements. This is done as follows:

\begin{verbatim}
Matrix(m, n, t) procedure Msum(A, B) ; Matrix(m, n, t) A, B ;
    begin integer i , j ;
    for i ← 1 step 1 until m do
    for j ← 1 step 1 until n do
        Msum[i, j] ← Add(t, A[i, j], B[i, j]) ;
    end ;

Matrix(m, n, t) procedure Mprod(A, B) ; Matrix(m, h, t) A; Matrix(h, n, t)B;
    begin integer i, j, k ; t temp ;
    for i ← 1 step 1 until m do
    for j ← 1 step 1 until n do
        begin temp ← zero(t) ;
        for k ← 1 step 1 until h do
            temp ← Add(t, temp, Mult(t, A[i, k], B[k, j]) ) ;
        Mprod[i, j ] ← temp ;
        end ;
    end ;
\end{verbatim}
Some uses of generalized matrix multiplication are as follows. If the mXm Boolean matrix \( A = (a_{ij}) \) has \( a_{ij} = \text{true} \) if there is a connection from node \( i \) to node \( j \) in the m-node directed graph \( G \) and has \( a_{ij} = \text{false} \) otherwise, and if \( \text{Add}(\text{Boolean}, x, y) = x \lor y \) and \( \text{Mult}(\text{Boolean}, x, y) = x \land y \) then the entry \( c_{ij} \) of the \( n^{\text{th}} \) power of \( A \) will be \text{true} if and only if there is a connection of \( n \) arcs from node \( i \) to node \( j \) in \( G \). Warshall [66] has given an algorithm which computes a matrix equivalent to the infinite sum of the matrices \( A^i \) (1 \( \leq i \leq \infty \)) whose entries tell if there exists a connection from node \( i \) to node \( j \) in the graph represented by the matrix \( A \). If the entries of the mXm matrix \( A \) contain lengths, times, costs, etc. to go from node \( i \) to node \( j \) in the \( a_{ij} \) position, and if \( \text{Add}(\text{cost}, x, y) = \min(x, y) \) and \( \text{Mult}(\text{cost}, x, y) = x + y \) then the general element of \( A^n \) is the cost (in distance, hours, dollars, etc) of the least costly path from node \( i \) to node \( j \) traversing \( n \) arcs or less.
8. A Programming Application:

Writing Recognizers for Graphical Objects Generated by Graphical Generative Grammars

Summary

This section gives a technique for writing recognizers for graphs that are generated from a class of graphical generative grammars. The recognizers are a generalization of Floyd-Evans production recognizers [22, 18] and may be used in a generalized FSL system [19, 20] to recognize graphical data as input for a compiler or an interpreter. Some examples of the kind of graphs that can be generated and recognized are (1) organic chemistry molecules, (2) formula trees and (3) simple geometric figures. The recognizers may be formulated to operate on graphs of 2, 3 or \( N \) dimensions, and may recognize objects independently of size, rotation and position in space. The grammars are a simple generalization of the phrase structure grammars for language strings. Data structures and an interpreter are given for a recognizer of 2-dimensional graphs with colored edges.

Graphical Generative Grammars

A production \( \alpha \rightarrow g \) is an ordered pair consisting of a
character $\alpha$ and a non empty graph $g$ which graph has the following properties: (1) $g$ is composed of a non empty set of characters (used as nodes) joined by colored edges (denoted in the sequel by straight line segments appropriately labelled) so as to form a connected graph, (2) there is one and only one occurrence of a dot ' . ' juxtaposed with one of the characters (or nodes) in $g$.

For example, $\beta \rightarrow .C \beta$ is a well formed production whose right hand side is a well formed connected graph (with edges of only one color) containing one and only one occurrence of a dot ' . ' juxtaposed with the node denoted by the character $C$. The graph with only one character and no edges is permitted so that $\alpha \rightarrow .m$ is an example of a well formed production.

Let $\alpha \rightarrow g$ be a production and let $h$ and $k$ be graphs such that $\alpha$ is a node of $h$ and such that $k$ is obtained from $h$ first by substituting $g$ for an occurrence of $\alpha$ in $h$ in such a way that all edges incident to $\alpha$ in $h$ before substitution are made incident to the dotted node in $g$ after substitution, and second by removing the dot in the result. If $k$ can be obtained from $h$ in this fashion then the relation $h \rightarrow k$ with respect to the production $\alpha \rightarrow g$ is said to hold.
For example, if $\beta \rightarrow .C-\beta$ is the production, and if $h$

$\text{H} \quad \text{H} \quad \text{H}$

is the graph $H-C-C-\beta$, then a derivative $k$ may be obtained from $h$ by replacing $\beta$ in $h$ with the graph $\begin{array}{c} .C-\beta \end{array}$ in such a way that all edges incident to $\beta$ in $h$ (of which there is one) are made incident to the dotted node in $\begin{array}{c} .C-\beta \end{array}$. This produces, upon substitution, the result $H-C-C-C-C-\beta$. The dot is then removed to produce the derivative $k$:

$k = H-C-C-C-C-\beta$

If the production had been $\beta \rightarrow .H$ then the result of substituting $\begin{array}{c} .H \end{array}$ for $\beta$ would have been $\begin{array}{c} .C-\beta \end{array}$.

Thus the position of the dot in $g$ is important in determining the result of the substitution.
Let $P$ be a set of graphical productions. The relation $h \Rightarrow k$ holds with respect to $P$ if there is a finite sequence $h_0, h_1, \ldots, h_n$ where $h = h_0$ and $k = h_n$ such that $h_{i-1} \rightarrow h_i$ for each $i$ in the range $1 \leq i \leq n$ with respect to some production chosen from $P$. Here $k$ is said to be a graphical derivative of $h$ with respect to $P$.

Let $G$ be a set of productions containing exactly one character $\gamma$ which appears only on the left of "$ightarrow$", let $T$ be a set of (terminal) characters which appear only on the right of "$ightarrow$" (represented by upper case Latin letters in the sequel), and let $V-T$ be the set of (non-terminal) characters which appear both on the left and on the right of "$ightarrow$" (represented by Greek letters in the sequel).

The graphical derivatives of $\gamma$ with respect to $G$ are called graphical forms of $\gamma$ with respect to $G$ and those graphical forms whose characters or nodes belong only to $T$ are terminal graphical forms, or terminal graphs for short. The set $L$ of terminal graphs with respect to a given set of graphical productions $G$ may be termed a graphical language, and $G$ a graphical phrase structure grammar.†

† A linearized notation may be given for graphical forms permitting them to be represented and manipulated as expressions in computers; in fact, some form of linearization is always forced when representing graphs in computers with linear memories.
Without loss of generality we may assume that for each graphical language \( L \) and associated grammar \( G \), each non-terminal character used in the productions of \( G \) is used in the derivation of at least one terminal graph of \( L \). Otherwise we may discard the productions containing unused non-terminals since they do not participate in the derivations of terminal graphs in \( L \) and this discard will have no effect on the generative power of \( G \).

As an example, the following graphical phrase structure grammar generates written two dimensional representations of single chain saturated hydrocarbon molecules.

\[
\alpha \rightarrow .H-\beta \\
\beta \rightarrow .C-H \\
\beta \rightarrow .C-\beta 
\]

An example of a derivation (of ethane) is

\[
\alpha \rightarrow H-\beta \rightarrow H-C-\beta \rightarrow H-C-C-H
\]
These graphical phrase structure grammars are not perfectly general. For example, we cannot derive the graph

\[
\begin{array}{c}
\text{X} \\
\text{X} \quad \text{X}
\end{array}
\]

from \( \alpha \) and \( \alpha \to \begin{array}{c}
\text{X} \\
\text{X} \quad \text{X}
\end{array} \) but can derive only \( \begin{array}{c}
\text{X} \\
\text{X} \quad \text{X}
\end{array} \). That is, edges incident to a non-terminal cannot be made incident to more than one node of the graph substituted for the non-terminal as we might desire in a more general generative capability (cf. Narasimhan [42]).

It is not clear that we are seriously restricted by this inability, however. For example, the graph

\[
\begin{array}{c}
\text{X} \\
\text{X} \quad \text{X}
\end{array}
\]

can be generated by the following grammar:

\[
\begin{align*}
\alpha & \to \begin{array}{c}
\text{X} \\
\text{X} \quad \text{X}
\end{array} \\
\mu & \to \begin{array}{c}
\text{X}
\end{array} \\
\pi & \to \begin{array}{c}
\text{X}
\end{array} \\
\rho & \to \begin{array}{c}
\text{X}
\end{array}
\end{align*}
\]
Graphical Reduction Recognizers

A reduction has the following format

\[ \text{LABEL} \rightarrow \text{LEFT GRAPH} | \rightarrow \text{RIGHT GRAPH} | \text{ACTION} \text{ LINK} \]

where the appearance of the LABEL, the arrow \( \rightarrow \) and the ACTION is optional. The left and right graphs are constructed from nodes, colored edges and a dot with the following restrictions:

1. Nodes may be either
   - (a) Terminal Characters,
   - (b) Non-Terminal Characters,
   - (c) Meta Characters, i.e. names of sets of characters, or
   - (d) The special meta character \( \sigma \) which stands for the universal set of characters,
   - (e) And nodes may bear integer subscripts.

2. Edges are straight line segments (of different colors if necessary).

3. There is one and only one occurrence of a dot \( . \) juxtaposed with one of the nodes.

4. The graph is connected.

Some examples of reductions are:

\[ \beta_1 \]
\[ H \]
\[ \beta \]
\[ \rightarrow \]
\[ . \beta \]
\[ \text{EXEC} 2 \]
\[ \beta_1 \]

(1)

\[ .H \rightarrow C \]
\[ H \rightarrow .C \]

\[ \beta_0 \]

(2)
The reduction (1) above causes an alteration of its left graph to its right graph. The arrow "\rightarrow" appears when an alteration is performed. The reduction (2) moves the location of the dot from juxtaposition with H in the left graph to juxtaposition with C in the right graph. No arrow appears when the dot, hereafter called the locus, is being moved. When a reduction is used to move the locus, the left and right graphs must be identical except for the locus. Note also that reduction (1) has an action EXEC 2 and reduction (2) has neither an action, nor a label, nor an arrow. This exemplifies the optional presence of the action, the label and the arrow.

To explain the operation of the reductions we choose a specific simple example. Let L be the graphical language of all written representations of linear straight chain saturated hydrocarbons, e.g.

\[
\begin{align*}
\text{H} & \text{H} \text{H} \text{H} \text{H} \\
\text{H} & \text{C} \text{H} & \text{C} \text{C} \text{H} & \text{C} \text{C} \text{H} & \text{C} \text{C} \text{H} & \text{C} \text{C} \text{H} & \text{C} \text{C} \text{H} & \text{C} \text{C} \text{H}
\end{align*}
\]

Let the grammar G consist of the following productions

\[
\alpha \rightarrow \text{H} \beta
\]

\[
\beta \rightarrow \text{C} \text{H}
\]

\[
\beta \rightarrow \text{C} \beta
\]
We see that $L$ consists of the terminal graphical forms of $\alpha$ with respect to $G$. Let $R$ be the following ordered sequence of reductions:

\[
\begin{align*}
\alpha_0 & : H - C & | & H - .C & | & \beta_0 \\
\beta_0 & : C - C & | & C - .C & | & \beta_0 \\
\beta_1 & : C - \beta & | & \beta & | & \beta_1 \\
\beta_1 & : H - .C & | & .C - H & | & \beta_1 \\
\end{align*}
\]

Choose now the ethane molecule derived on page 141 together with a dot juxtaposed with its left most hydrogen atom.

Call this ethane molecule $g$. We enter the recognizer $R$ at the reduction labelled $\alpha_0$. To apply a reduction

\[
\begin{align*}
\alpha_0 & : H - C & | & H - .C & | & \beta_0 \\
\end{align*}
\]
to g we superimpose the locus of the left hand graph on the locus of g and determine if the left hand graph matches a subgraph of g with the loci thus superimposed. If so we say that the reduction applies to g and the subgraph of g that matches is transformed to the right graph of the reduction. This is done either by moving the locus in g to the node specified in the right graph (in the case no arrow appears in the reduction), or by replacing the subgraph matching the left graph by the right graph (in the case that an arrow does appear in the reduction). The rules involving graphs with meta characters or σ's as nodes are similar. If a locus is moved to or from a meta character or σ it is moved to or from the node in g that matches the meta character or σ. A meta character or σ matches a node if the character at that node is a member of the class of characters named by the meta character or σ. If a meta character or σ is used in both a left graph and a right graph the nodes matching the meta characters or σ's in the left graph are used when the corresponding meta characters or σ's appear in the construction of the right graph. Where ambiguity arises in this correspondence subscripts must be used to specify the proper correspondence.

For example, in the reduction

\[
X \quad \sigma \quad \leftarrow \quad \sigma_2 \rightarrow \sigma \rightarrow \sigma_1 \quad Y
\]

\[
\sigma_1 \quad \sigma_2
\]
if the left graph matches then the node matching $\sigma_2$ is used to the left of the node matching $\sigma$ in constructing the right graph to be substituted.

Hence the left graph of the reduction

$$\alpha_0 \quad .H\longrightarrow C \mid \quad H\longrightarrow C \quad \mid \quad \beta_0$$

matches the subgraph of $g$ indicated by enclosure within dotted lines in the following diagram

causing upon replacement of the left graph by the right graph, a shift of locus from the leftmost $H$ to the $C$ immediately to its right. The result is

When a reduction applies control passes to the reduction whose label is given by its link. In this case, control passes to the reduction labelled $\beta_0$. In the event that a reduction does not apply control passes to the next reduction in sequence. The reduction at $\beta_0$ applies since its left graph $\quad .C\longrightarrow C \quad$ matches the subgraph of
enclosed by dotted lines. This causes, upon replacement of the left graph by the right graph, another shift in locus producing as a result the graph

\[ \text{H} \text{- C - C - H} \]

Control then returns to the reduction \( \beta_0 \). This time the reduction at \( \beta_0 \) fails to apply since its left graph

\[ \text{H} \text{- C - C} \]

is not superimposable as a subgraph on

\[ \text{H} \text{- C - C - H} \]

when the loci are superimposed, so control passes to the next reduction in sequence, which reduction, lacking a label, may be called "the reduction at \( \beta_0+1 \)", meaning that it is the first reduction in sequence after the one labelled \( \beta_0 \). This reduction applies since its left graph matches the subgraph enclosed in dotted lines in

\[ g \text{ as follows: } \]

is then replaced by the right graph \( \beta \) transforming \( g \) into

Control now passes to the reduction at \( \beta_1 \). The left graph at \( \beta_1 \) matches
the subgraph enclosed in dotted lines in $g$ as follows: 

and this matched subgraph is replaced by the right graph of $\beta 1$
yielding $H - \beta$ as a result. Control then returns to $\beta 1$. The
reduction at $\beta 1$ now fails to apply and control passes to the reduction
at $\beta 1 + 1$ which applies and effects a final reduction of $H - \beta$ to $\alpha$.
An exit is made from the recognizer $R$ at this point with the task
complete. This reduction sequence can be signified more
compactly in the following notation:

$$H \stackrel{\alpha 0}{\longrightarrow} H \rightarrow H \stackrel{\beta 0}{\longrightarrow} H \rightarrow H \stackrel{\beta 0 + 1}{\longrightarrow} H \rightarrow H \stackrel{\beta 1}{\longrightarrow} H \rightarrow H \stackrel{\beta 1 + 1}{\longrightarrow} \alpha$$

In this notation the symbols above each right arrow signify which
reduction in $R$ applied to transform the graph on the left of the arrow
into the graph on the right of the arrow.

Adding Semantics to Graphical Recognizers

We wish now to show how to couple graphical reduction
recognizers with semantic routines in such a way that the recognizers
can be used to drive interpreters or compilers accepting graphical inputs. To do this we generalize the FSL model \[19, 20\]. Thus we have a series of numbered EXEC routines which may appear in the action fields of reductions. When a reduction containing a numbered EXEC routine in its action field applied to an input graph \( g \) a transfer of control is made to this numbered EXEC routine. The EXEC routine can refer to and manipulate descriptive data structures associated with nodes of \( g \). In the FSL model, the variables \( \text{left } 1, \text{left } 2, \ldots, \text{left } N \) refer to descriptive structures associated with elements of a stack matching the left hand side of a reduction before the transformation associated with the reduction takes place, and the variables \( \text{right } 1, \text{right } 2, \ldots, \text{right } N \) refer to descriptive structures associated with elements of a transformed stack matching the right hand side of the reduction after the reduction has been applied.

In our graphical generalization, a correspondence between descriptive data structures and FSL variables of the form \( \text{left } K \) and \( \text{right } K \) is set up by subscripting the nodes of the left and right graph respectively. If \( \alpha_N \) is a subscripted node of a left graph \( L \) of a reduction which applies to an input graph \( g \), then \( \text{left } N \) refers to the descriptive data structure associated with the node of \( g \) matched by \( \alpha_N \). Similarly, if \( \beta_N \) is a subscripted node of a right graph \( R \) of a reduction which has been applied
to an input graph \( g \) producing a transformed graph \( g' \), then right \( N \) refers to the descriptive data structure associated with the node of \( g' \) matched by \( \beta_N \).

To write graphical FSL interpreters we permit the use of the data structure definition facility within EXEC routines. Hence, an interpreter for the language \( L \) of straight chain saturated hydrocarbons, which builds up interpretively an internal data structure for a straight chain saturated hydrocarbon input can be given. First assume the following data structure definitions hold.

```markdown
descriptor  hydrogen, carbon;

descriptor procedure  Atom(I,N); identifier I; integer N;

    Atom ← [name: (=I) | valence: (=N) | bondlist: N×any ] ;

hydrogen ← Atom(.H, 1) ;
carbon ← Atom(.C, 4) ;

The recognizer with subscripted nodes in its left and right graphs and with calls on the appropriate EXEC routines is given as follows:

\[
\begin{align*}
\alpha_1 & \quad .H \longrightarrow C \quad | \quad H \longrightarrow C \quad | \quad \beta_0 \\
\beta_0 & \quad \frac{H_1}{C_4} \longrightarrow H_2 \quad | \quad \rightarrow \quad .\beta_1 \quad | \quad \text{EXEC 1} \quad \beta_1
\end{align*}
\]
```
The EXEC routines to be coupled with this recognizer making the combination an interpreter are as follows:

EXEC 1

left 1 ← cons(hydrogen); left 2 ← cons(hydrogen);
left 3 ← cons(hydrogen); left 4 ← cons(carbon);
bondlist(left 4) ← cons(4Xany)(left 1, left 2, left 3,);
bondlist(left 1)[1] ← left 4; bondlist(left 2)[1] ← left 4;
bondlist(left 3)[1] ← left 4; right 1 ← left 4;

EXEC 2

left 1 ← cons(hydrogen); left 3 ← cons(hydrogen);
left 4 ← cons(carbon); bondlist(left 1)[1] ← left 4;
bondlist(left 3)[1] ← left 4; bondlist(left 2)[4] ← left 4;
bondlist(left 4) ← cons(4Xany)(left 1, left 2, left 3,);
right 1 ← left 4;

EXEC 3

left 1 ← cons(hydrogen); bondlist(left 1)[1] ← left 2;
bondlist(left 2)[4] ← left 1; right 1 ← left 2;
To change these EXEC's to make a compiler rather than an interpreter we would write the following:

EXEC 1

```
left 1 ← STORLOC↑\(^\dagger\) ATOM ; TALLY[STORLOC]↑
left 2 ← STORLOC \^ ATOM ; TALLY[STORLOC] ;
left 3 ← STORLOC \^ ATOM ; TALLY[STORLOC] ;
left 4 ← STORLOC \^ ATOM ; TALLY[STORLOC] ;
CODE(↑
    left 1 ← cons(hydrogen) ; left 2 ← cons(hydrogen) ;
    left 3 ← cons(hydrogen) ; left 4 ← cons(carbon) ;
    bondlist(left 4) ← cons( 4 \times any)(left 1, left 2, left 3,) ;
    bondlist(left 1)[1] ← left 4 ; bondlist(left 2)[1] ← left 4 ;
    bondlist(left 3)[1] ← left 4 ) ;
right 1 ← left 4 ;
```

EXEC 2

```
left 1 ← STORLOC \^ ATOM ; TALLY[STORLOC] ;
left 3 ← STORLOC \^ ATOM ; TALLY[STORLOC] ;
left 4 ← STORLOC \^ ATOM ; TALLY[STORLOC] ;
CODE(
    left 1 ← cons(hydrogen) ; bondlist(left 1)[1] ← left 2 ;
    bondlist(left 3)[1] ← left 4 ; bondlist(left 2)[4] ← left 4 ;
    bondlist(left 4) ← cons( 4 \times any)(left 1, left 2, left 3,) ) ;
right 1 ← left 4 ;
```

EXEC 3

```
left 1 ← STORLOC \^ ATOM ; TALLY[STORLOC ] ;
CODE (  
    left 1 ← cons(hydrogen) ; bondlist(left 1)[1] ← left 2 ;
    bondlist(left 2)[4] ← left 1 ) ;
right 1 ← left 2 ;
```

† See next page for explanation.
In FSL (cf. [20]) STORLOC is a pointer variable whose value at compile time is the address of a current storage location. TALLY[STORLOC] increments this pointer to the next storage location in sequence. At compile time we may associate storage addresses with variables. The result of compiling expressions involving the variables with which these addresses are associated is the creation of code that accesses, stores and acts on these addresses. This code is produced by operating on an expression E with the function CODE(E). In the above example, the meaning of the function CODE(E) must be extended to generate code for construction, selection and testing when E is an expression formed from cons, the notation for selectors (e.g. bondlist(left 2)[4]), or the notation for predicates (e.g. x == Atom(C, 4)).

The following pages present a series of three examples which demonstrate the types of recognizers that can be written with graphical reductions. The example presented in the text so far, while good for expository purposes, is constrained to a linear rather than a two dimensional scan and does not demonstrate properly the flexibility of the technique.

**Saturated Aliphatic Hydrocarbons**

The following grammar is a grammar for saturated aliphatic hydrocarbons:

```
In the following recognizer we agree to match graphs by similarity rather than by congruence. That is, the loci of the left graph and the input graph are superimposed, the nodes are compared, and if they match, all edges incident to the node with the locus in the left graph are matched against edges incident to the node with the locus in
the input graph. The edges match if they are of the same color and if they have the same direction regardless of their length. (We define this to be the meaning of similarity.) When edges match, the nodes at the ends are matched unless they have been processed before. Then all edges incident to the unprocessed node are processed unless these have been processed before. This process continues until the left graph either fails or succeeds in matching the input graph. Since all graphs are connected the above generation method is guaranteed to exhaust them during a successful match.

Meta Character \( X = \text{C or H} \).

\[
\begin{align*}
\alpha_0 & : \text{H--X} & | & \text{H--X} & | & \rho_0 \\
\rho_0 & : \text{C} & | & \text{C} & | & \tau_0 \\
\tau_0 & : \text{X--C} & | & \text{X--C} & | & \lambda_0 \\
\lambda_0 & : \text{C} & | & \text{C} & | & \beta_0 \\
\beta_0 & : \text{H} & | & \text{H} & | & \eta_1 \text{--C} & | & \lambda_1 \text{--C} & | & \text{EXEC 1} \lambda_1
\end{align*}
\]
\[
\begin{align*}
\beta_0 & \quad .C & - X & \quad | & \quad C & - .X & \quad | & \quad \rho_0 \\
& \quad .H_1 & \quad | & \quad \beta_1 & \quad \text{EXEC 1} & \quad \beta_1 \\
\rho_1 & \quad \lambda_1 & - .C_4 & - \rho_3 & \quad | & \quad .T_1 & \quad | & \quad \text{EXEC 2} & \quad \tau_1 \\
& \quad X & \quad | & \quad X & \quad \text{EXEC 3} & \quad \text{EXIT} \\
& \quad \alpha_1 & \quad \beta_0 & \quad \beta_0 \\
\tau_1 & \quad \lambda_1 & - .C_4 & - X & \quad | & \quad .T_1 & \quad | & \quad \text{EXEC 2} & \quad \lambda_1 \\
& \quad \beta_1 & \quad \beta_1 \\
& \quad \beta_3 & \quad \beta_3 \\
\beta_1 & \quad X & - .C_4 & - \rho_2 & \quad | & \quad X & - \rho_1 & \quad \text{EXEC 2} & \quad \rho_1 \\
& \quad \beta_2 & \quad \beta_2 \\
& \quad \beta & \quad \beta \\
\lambda_1 & \quad .\lambda_3 & - .C_4 & - \rho_1 & \quad | & \quad .T_1 & \quad | & \quad \text{EXEC 2} & \quad \beta_1 \\
& \quad \beta_2 & \quad \beta_2 \\
& \quad \lambda & - C & \quad | & \quad \lambda & - C & \quad \tau_0 \\
\end{align*}
\]
The semantics for an interpreter that builds internal representations of hydrocarbon molecules from this recognizer are as follows:

EXEC 1  left 1 ← cons(hydrogen) ; right 1 ← left 1 ;

EXEC 2  left 4 ← cons( carbon ) ; bondlist(left 4) ← cons( 4 × any ) (left 1, left 2, left 3 , ) ;
        if name(left 1) = . C then bondlist(left 1)[4] ← left 4  else
           bondlist(left 1)[1] ← left 4 ;
        if name(left 2) = . C then bondlist(left 2)[4] ← left 4  else
           bondlist(left 2)[1] ← left 4 ;
        if name(left 3) = . C then bondlist(left 3)[4] ← left 4  else
           bondlist(left 3)[1] ← left 4 ;
        right 1 ← left 4 ;

EXEC 3  left 1 ← cons(hydrogen) ; bondlist(left 1)[1] ← left 2 ;
        bondlist(left 2)[4] ← left 1 ; right 1 ← left 2 ;

The following sequence of reductions demonstrates a typical graphical analysis performed by the recognizer. We have chosen to use
2, 3-methyl propane, normally displayed as

```
    H   C   H
   /     /     \
  H---C---H
     /   /   \
    H   H   H
```

but displayed in the following example in a somewhat non-standard configuration, in order to show the versatility of the recognizer.
We might add at this point that more elaborate forms of recognizers can recognize larger families of hydrocarbons using edges of different color for double and triple bonds and recognizing ring structured compounds. Space precludes giving examples of these.

**Formula Trees**

The following recognizer recognizes formula trees constructed from binary and unary operators. There are three meta characters: atom, binop, and unop, defined as follows:

\[
\begin{align*}
\text{atom} & = \{a, b, c, \ldots, z\} \\
\text{binop} & = \{+, - , \times , \div\} \\
\text{unop} & = \{\sin, \cos, \tan, \ln, \exp, - \}
\end{align*}
\]

The grammar is as follows:

\[
\begin{align*}
\text{formula} & \rightarrow \text{binop} \\
& \quad \quad \quad \quad \text{formula} \\
& \quad \quad \quad \quad \text{formula} \\
\text{formula} & \rightarrow \text{unop} \\
& \quad \quad \quad \quad \text{formula} \\
\text{formula} & \rightarrow \text{atom}
\end{align*}
\]

The recognizer is given by the following set of reductions:
\[
\begin{align*}
\text{f} & \quad \text{binop} & | & \rightarrow & \quad \text{formula} & | & \quad \text{f} \\
\quad & \quad \text{formula} & \quad & \text{formula} \\
\text{+1} & \quad \text{unop} & | & \rightarrow & \quad \text{formula} & | & \quad \text{f} \\
\quad & \quad \text{formula} \\
\text{+2} & \quad \text{binop} & | & \quad \text{binop} & \quad \text{formula} & \quad \text{f} \\
\quad & \quad \text{formula} & \quad \sigma \\
\text{+3} & \quad \text{unop} & | & \quad \text{unop} & \quad \sigma \\
\quad & \quad \sigma \\
\text{+4} & \quad \text{binop} & | & \quad \text{binop} & \quad \sigma \\
\quad & \quad \sigma \\
\text{+5} & \quad \sigma & | & \rightarrow & \quad \sigma & | & \quad \text{f} \\
\quad & \quad \text{atom} & \quad \text{formula} \\
\text{+6} & \quad \sigma & | & \rightarrow & \quad \sigma & | & \quad \text{f} \\
\quad & \quad \text{formula} & \quad \text{formula} \\
\text{+7} & \quad \sigma & | & \rightarrow & \quad \sigma & | & \quad \text{f} \\
\quad & \quad \text{atom} & \quad \text{formula} \\
\text{+8} & \quad \sigma & | & \rightarrow & \quad \sigma & | & \quad \text{f} \\
\quad & \quad \text{formula} & \quad \text{formula} \\
\text{+9} & \quad \sigma & | & \rightarrow & \quad \sigma & | & \quad \text{f} \\
\quad & \quad \text{atom} & \quad \text{formula} \\
\text{+10} & \quad \sigma & | & \rightarrow & \quad \sigma & | & \quad \text{f} \\
\quad & \quad \text{formula} & \quad \text{formula} \\
\text{+11} & \quad \text{formula} & | & \rightarrow & \quad \text{formula} & | & \quad \text{exit}
\end{align*}
\]
The following is an example of a series of reductions using this formula tree recognizer:
Geometric Figures

The following grammar generates a geometric figure.
If we agree not to display nodes and to extend edges to touch the defined locations for the nodes the figure becomes closed. This is easy to arrange on certain graphical displays [51].

\[
\begin{align*}
\alpha & \rightarrow \quad \text{\begin{tikzpicture}[scale=0.5]
\node (a) at (0,0) {$\nu$};
\node (b) at (1,1) {$\pi$};
\node (c) at (2,0) {$\rho$};
\draw (a) -- (b) -- (c);
\end{tikzpicture}} \\
\nu & \rightarrow \quad \text{\begin{tikzpicture}[scale=0.5]
\node (a) at (0,0) {$\delta$};
\node (b) at (1,1) {$\mu$};
\node (c) at (0.5,0.5) {$\epsilon$};
\draw (a) -- (b) -- (c);
\end{tikzpicture}} \\
\pi & \rightarrow \quad \text{\begin{tikzpicture}[scale=0.5]
\node (a) at (0,0) {$a$};
\node (b) at (1,1) {$a$};
\node (c) at (2,0) {$b$};
\draw (a) -- (b) -- (c);
\end{tikzpicture}} \\
\rho & \rightarrow \quad \text{\begin{tikzpicture}[scale=0.5]
\node (a) at (0,0) {$d$};
\node (b) at (1,1) {$c$};
\draw (a) -- (b);
\end{tikzpicture}} \\
\epsilon & \rightarrow \quad \text{\begin{tikzpicture}[scale=0.5]
\node (a) at (0,0) {$m$};
\node (b) at (1,1) {$m$};
\draw (a) -- (b);
\end{tikzpicture}} \\
\delta & \rightarrow \quad \text{\begin{tikzpicture}[scale=0.5]
\node (a) at (0,0) {$b$};
\node (b) at (1,1) {$b$};
\draw (a) -- (b);
\end{tikzpicture}} \\
\mu & \rightarrow \quad \text{\begin{tikzpicture}[scale=0.5]
\node (a) at (0,0) {$x$};
\end{tikzpicture}}
\end{align*}
\]
The figure that $\alpha$ generates is given with nodes displayed as follows:

Without nodes displayed, this figure appears as follows:

A recognizer for these figures is given by the following set of reductions:
A series of reductions using this recognizer proceeds as follows:
An Interpreter for Reductions

In this section we exhibit a set of data definitions and a program defining an interpreter for the recognition of a graph \( g \) with respect to a set of reductions \( R \). Recognition is by congruence in two dimensions of graphs with colored edges. The following set of definitions define the data of interest:

Data Structures

descriptor graph, realpair, node, edge, subscriptnode, designator, metacharacter, reductiongraph, ReductionSet, Reduction;

\[
\text{graph} \leftarrow [\text{locus: realpair} | \text{nodelist: List(node)} | \text{edgelist:List(edge)}];
\]

\[
\text{realpair} \leftarrow [x:real | y:real];
\]

\[
\text{node} \leftarrow [\text{name: identifier} | x:real | y:real | \text{semantics: any} | \text{mark:Boolean}];
\]

\[
\text{edge} \leftarrow [x:real | y:real | \text{deltax:real} | \text{deltay:real} | \text{color: identifier} | \text{mark:Boolean} |
\hspace{1cm} \text{correspondent: edge}];
\]

\[
\text{subscriptnode} \leftarrow [\text{name: designator} | \text{subscript: integer} | x:real | y:real |
\hspace{1cm} \text{correspondent: node}];
\]

\[
\text{designator} \leftarrow \text{metacharacter} \triangleright [\text{identifier}] \triangleright \{\text{sigma}\};
\]

\[
\text{metacharacter} \leftarrow \text{List(identifier)};
\]

\[
\text{reductiongraph} \leftarrow [\text{locus: realpair} | \text{nodelist: List(subscriptnode)} | \text{edgelist: List(edge)}];
\]

\[
\text{ReductionSet} \leftarrow \text{List(Reduction)};
\]

\[
\text{Reduction} \leftarrow [\text{label: identifier} | \text{LeftGraph: reductiongraph} | \text{arrow:Boolean} | \text{RightGraph: reductiongraph} | \text{action: formula} |
\hspace{1cm} \text{link: identifier}];
\]

The Program

In the following set of procedures, the procedure Interpret(\( R, g \))
defines an interpreter which acts on a set of reductions $R$ to recognize a graph $g$. We assume that $R$ is a global variable of type List containing a list of reductions that constitute a recognizer, and that $g$ is a global variable of type graph containing a graph to be recognized. We assume that during the construction of $g$ the mark component of nodes and edges has been initialized to `false`, and that there is a global variable, `currentreduction`, of type Reduction set initially to $\text{Head}(R)$, the first reduction in $R$.

The variables $R$, $g$ and `currentreduction` provide communication across the procedure `Interpret(R, g)` and its associated auxiliary procedures.

```plaintext
procedure Interpret(R, g); List(any) R; graph g;
begin Reduction currentreduction, temp; List L; L ← R;
loop1: if Apply(currentreduction, g) then
    begin eval (action (currentreduction));
    go to findlink; end
else if ~ Nullist(currentreduction) then
    begin currentreduction ← Tail(R)[1]; L ← Tail(R);
    go to loop1; end
    begin Print('input graph is ungrammatical');
    go to exit; end
findlink: temp ← R;
loop2: if link(currentreduction) = label(Head(temp)) then
    begin L ← temp; currentreduction ← Head(L);
    go to loop1; end
else
    begin temp ← Tail(temp); go to loop2; end;
exit:
end;
```
Auxiliary Procedures for the Interpreter

Boolean Procedure Apply(currentreduction, g) ; Reduction currentreduction ;
    graph g ;
    begin
    Apply ← Subgraph( leftgraph(currentreduction) , g ) ;
    if Apply then replace(leftgraph(currentreduction),
        rightgraph(currentreduction), g ) ;
    end ;

    comment: The procedure Apply(c, g) determines whether the current
    reduction c applies to the input graph g and if so it transforms
    g according to the transformation expressed by the current
    reduction ;

Boolean Procedure Subgraph (A,B) ; graph A,B ;
    Subgraph ← Subset(nodelist(A), nodelist(B) ) ^
        Subset(edgelist(A), edgelist(B) ) ;

Boolean Procedure Subset(L, K) ; List(any) L,K ;
    Subset ← if Nullist(L) then true else
        Member2( Head(L), K ) ^ Subset(Tail(L), K) ;

Boolean Procedure Member2(x,K) ; any x ; List K ;
    Member2 ← if Nullist(K) then false else
        Match( x, Head(K) ) v Member2(x, Tail(K)) ;

    comment: The following procedure, Match(a, b), matches either a
    subscriptnode in the left graph against a node in the input graph or
    an edge in the left graph against an edge in the input graph. If a match
    occurs, then, as a side effect, the node or edge in the input graph
    is stored as the correspondent component of its matching subscriptnode
or edge in order that after transformation by an applicable reduction an access path will exist permitting access to the current values of the variables of the form left K which values are stored as the semantics components of nodes;

Boolean Procedure Match(a, b); any a, b;
begin
if subscriptnode(a) ^ node(b) then
begin
  Boolean nametest, nodetest;
  nametest ← if name(a) == 'metacharacter then
    member ( name(a), name(b) ) else
    if name(a) = 'sigma' then true else
    name(a) = name(b);
  nodetest ← if nametest then
    x(a) + x(locus(g)) ^
    y(a) + y(locus(g)) = y(b) else false;
  Match ← nodetest;
  if nodetest then begin correspondent(a) ← b ;
    mark(correspondent(a)) ← true end;
end;
if edge(a) ^ edge(b) then
begin
  Match ← x(a) + x(locus(g)) = x(b) ^
  y(a) + y(locus(g)) = y(b) ^
  deltay(a) = deltay(b) ^
  color(a) = color(b);
  if Match then begin correspondent(a) ← b ;
    mark(correspondent(a)) ← true ; end;
end;

comment: Here we see that the coordinates of a left graph are always relative to a locus of (0, 0) for all nodes and edges in a left graph, and that to compare coordinates of nodes and edges with those of the input graph the coordinates of the locus of the input graph must be added;
The following procedure replaces an instance of the
subgraph $a$ by the subgraph $b$ in the input graph $g$. It is
therefore responsible for carrying out the transformation
associated with a reduction. We note that for arrow reductions
locus(b) = (0, 0) and for non arrow reductions locus(b) = ($\Delta x, \Delta y$)
the vector increment needed to move the locus;

Procedure Replace (a, b, g); reductiongraph a, b; graph g;
begin
  $x(locus(g)) \leftarrow x(locus(g)) + x(locus(b))$;
  $y(locus(g)) \leftarrow y(locus(g)) + y(locus(b))$;
  if arrow(currentreduction) = false then go to exit;
  GraphDelete(a, g); GraphInsert(b, g);
exit:    ;
end;

Procedure GraphDelete(a, g); reductiongraph a; graph g;
begin
  nodelist(g) ← Reduce(nodelist(g));
  edgelist(g) ← Reduce(edgelist(g)); end

List Procedure Reduce(L); List(any) L;
Reduce ← if Nullist(L) then L else
    if mark(Head(L)) then Reduce(Tail(L)) else
    cons(Pair(any))(Head(L), Reduce(Tail(L)));

Procedure GraphInsert(b, g); reductiongraph b; graph g;
begin
  List(any) L, K; node n; edge e;
  K ← edgelist(b); L ← cons(Nullist);
loop1: if Nullist(K) then go to L1; 
    e ← cons(edge)( x(Head(K)) + x(locus(g)) ,  
                        y(Head(K)) + y(locus(g)) ,  
                        deltax(Head(K)),  
                        deltay(Head(K)),  
                        color(Head(K)),  
                        false, nil );  
    L ← cons(Pair(any))(e, L );  K ← Tail(K); go to loop1;  

L1: edgelist(g) ← concatenate(L, edgelist(g));  
    L ← cons(Nullist);  K ← nodelist(b);  

loop2: if Nullist(K) go to L2;  
    n ← cons(node)(getname(Head(K)), x(Head(K)) + x(locus(g)) ,  
                        y(Head(K)) + y(locus(g)), nil, false );  
    L ← cons(Pair(any))(n, L );  K ← Tail(K); go to loop2;  

L2: nodelist(g) ← concatenate(L, nodelist(g));  

end procedure ;

identifier procedure getname(n); subscriptnode n;  
    if name(n) == "metacharacter v name(n) = 'sigma' then  
        getname ← findcorrespondent(leftgraph(currentreduction),  
                                name(n), subscript(n) );  
    else getname ← name(n);  

identifier procedure findcorrespondent(g, n, s); reductiongraph g;  
    designator n; integer s;  
    begin List(any) L;  L ← nodelist(g);  
    loop: if Nullist(L) then begin Print('error no correspondent' )  
        go to exit; end ;
if name(Head(L)) = n \land subscript(Head(L)) = s \text{ then}
\begin{align*}
\text{begin} & \hspace{1em} \text{findcorrespondent} \leftarrow \text{name(correspondent(Head(L)))} ; \\
& \hspace{1em} \text{go to exit; end} ; \\
\text{else} & \hspace{1em} L \leftarrow \text{Tail(L)} ; \hspace{1em} \text{go to loop} ;
\end{align*}

exit \hspace{1em} ;
\end{align*}

This completes the set of auxiliary procedures for the reduction interpreter.

Generalizations

One can see that it is a straightforward extension of the above set of procedures to include a third or z-coordinate in all the processes that treat coordinates of nodes, edges and loci. This extension would allow the recognizer to operate on graphical data structures in three dimensions. A corresponding addition of a z-coordinate could then be made to all underlying data structures utilizing coordinate information. Additions to data structure definitions and procedures permitting N-coordinates instead of two or three would permit graphs in N-dimensions to be recognized. Sonic wands could be used to construct three dimensional graphs (the sonic wand being L.G. Roberts' three dimensional equivalent of a light pen), and stereo-sopic binocular projection systems could be used to display three dimensional objects.
By arranging for the coordinates of nodes and edges in left and right graphs of reductions to be relative to a (0, 0) locus, superimposition of left and right graphs on the input graph is performed by adding the coordinates of the locus of the input graph to the relative coordinates of nodes and edges in the left and right graphs. Thus recognition of input graphs is independent of translation. For independence of magnification and contraction graphs must be matched by similarity. The procedures above match by congruence only and do not generalize readily to matching by similarity. To match by similarity the matching of edges must be made with regard to edge direction but not edge length (a simple matter to arrange by means of ratios of increments of coordinates such as \( \frac{\Delta x_1}{\Delta y_1} = \frac{\Delta x_2}{\Delta y_2} \) for \( \Delta y_1 \) and \( \Delta y_2 \neq 0 \)), and the search technique used to exhaust the graph must be made more elaborate (cf. page 155 bottom).

For independence of rotation in two dimensions one can make the graphs in the first set of reductions to be matched assymetrical about the locus (by agreeing that the character for a node adjacent to the locus shall be positioned at a certain bearing from that locus). When any of these first reductions are interpreted by attempting to match their left graphs against the input graph a sequence of rotations through discrete angular increments is performed and a match is attempted at each step. Two edges match if their directions match within the tolerance of an
angular increment. Once the first production is matched the angular increment needed to make it match is applied to all graphs of all reductions. The remainder of the recognition proceeds as before. Thus the angle of rotation is determined upon matching the first reduction and is used in applying the rest. Independence of translation, magnification and rotation makes the recognition method invariant under affine transformations of the input graph.

For graphs, such as organic molecules involving double and triple bonds, colored edges are used to represent different bond types.

Work of Others

Anderson [2] has given a method based on adjacency relations between rectangles for scanning and recognizing arithmetic expressions displayed in two dimensions. His technique is more effective than the graphical technique presented above for recognizing the class of two dimensional representations of arithmetic expressions. Other work of interest on two dimensional recognition techniques has been done by Klerer and May [30], Narasimhan [42] and Wells [68].
9. Organic Chemistry Molecules

Exciting work has been done recently by Joshua Lederberg of Stanford on a system called 'Dendral' \[35, 36\] which can compute, among other things, all irredundant topological combinations of a set of atoms which satisfy the valence requirements of those atoms. A main contribution of Dendral is that it supplies a cannonical notation for organic molecules. This notation may be represented by the use of unary, binary and ternary prefix operators which operate on the chemical symbols for atoms and on prefix expressions recursively to represent molecules which are trees. For molecules containing cyclic structures, Hamiltonian circuits through the molecule provide the key to giving a canonical form.

The presentation given in this section gives a procedure for finding a chemical name for representations of molecules in the class of saturated aliphatic hydrocarbons. While infix formulae (somewhat similar to Lederberg's prefix formulae) are used to represent tree structured hydrocarbons, the object of the presentation differs from Lederberg's in that it attempts
to compute names, and not to generate exhaustively or selectively a class of topological structures representing organic molecules.

We begin by giving a naming system for hydrocarbons.

**Names for Straight Chain Hydrocarbons**

Straight chain hydrocarbons with no side chains constructed from single bonds are members of the "Alkane" series of hydrocarbons: \( \text{CH}_4, \text{C}_2\text{H}_6, \text{C}_3\text{H}_8, \ldots, \text{C}_n\text{H}_{2n+2} \). The names of the first ten members of the Alkane series are: methane, ethane, propane, butane, pentane, hexane, heptane, octane, nonane and decane.

Their two dimensional written representations are generated by the graphical grammar given on page 141 of Section 8. Examples of these two dimensional written representations are:

- \( \text{H} - \text{C} - \text{H} \)
- \( \text{H} - \text{C} - \text{C} - \text{H} \)
- \( \text{H} - \text{C} - \text{C} - \text{C} - \text{H} \)

methane  ethane  propane

Side chains of a molecule in the Alkane series may be attached by replacing one of its side bonds to a hydrogen atom \( \text{H} \) with a bond to an Alkane radical of the form \( -(\text{CH}_2)_k\text{CH}_3 \) where \( k = 0, 1, 2, \ldots \).

An example of a molecule with a side chain is the following:
The chemical name for this molecule is 5-ethyl nonane, since a side chain -CH₂CH₃ consisting of the ethyl radical is attached to a nonane molecule at its fifth carbon atom. Carbon atoms are numbered from left to right along the longest chain of carbon atoms in the molecule. The length of the longest chain is used to provide the name for the basic molecule. If the length is six, it is a hexane; if the length is seven, it is a heptane; if the length is eight, it is an octane; and so on. Side chains are given names according to their lengths also. If a side chain contains four carbon atoms in a chain it is a butyl side chain; if a side chain contains five carbon atoms in a chain it is a pentyl side chain, and in general, the name for the side chain is derived from the name for the corresponding Alkane of the same length by substituting the suffix 'yl' for the suffix 'ane' in the Alkane name. Hence, straight chain radicals derived from Alkanes are called Alkyl radicals. Side chains may of course have side chains, and recursively, these may have side chains, and so on. Of course, more than one side chain may be attached to an Alkane molecule. In this case, the
adjectives for the several side chains are listed in order
prefixed by a number indicating the carbon atom on the longest
chain to which they are affixed. E.g.

\[
\begin{align*}
&\text{H} \\
&\text{H} - \text{C} - \text{H} \\
&\text{H} \quad \text{H} \quad \text{H} \quad \text{H} \quad \text{H} \quad \text{H} \\
&\text{H} - \text{C} - \text{C} - \text{C} - \text{C} - \text{C} - \text{C} - \text{C} - \text{C} - \text{H} \\
&\text{H} \quad \text{H} \quad \text{H} \quad \text{H} \\
&\text{H} - \text{C} - \text{H} \\
&\text{H} - \text{C} - \text{H} \\
&\text{H}
\end{align*}
\]

could be named 3-methyl, 5-ethyl octane. Other more elaborated naming conventions resolve ambiguous ways of naming compounds (for example, an alternative name for the above compound is 4-ethyl, 6-methyl octane if we number carbon atoms from the right to the left). These conventions are given in the Geneva naming system (see reference [25] pages 666-683). However, for illustrative purposes and for simplicity, we shall confine ourselves to the simple naming system introduced above.

We shall assume that a graphical recognizer, constructed according to the method given in Section 8, drives a set of construction routines which build up internal representations of saturated branching
hydrocarbons from data structures of the following sort:

```
descriptor procedure Atom(N, I); integer N; identifier I;
    Atom ← [name: (=I) | valence: (=N) | bondlist: N × any |
           mark: Boolean ];
carbon ← Atom(4, C);
hydrogen ← Atom(1, H);
```

The recognizer on pages 156 - 157 is sufficient to accomplish these constructions provided we extend the construction routines to initialize the 'mark' components of all constructed carbon and hydrogen atoms to the value `false`. The mark component will be used to prevent redundant scanning of the data structures. When scanned, structures whose mark is changed from `false` to `true` are not processed. After a structure is processed its mark is changed from `false` to `true` to prevent reprocessing.

The process of finding a name for a branched hydrocarbon consists first in finding the longest chain and then in stepping along it calculating recursively the names of the side chains. Matrix multiplication may be used to find the longest chain of carbon atoms (which may be non-unique) by means of a multiplication on paths in a graph. The elements of the matrix are paths along carbon chains
and path multiplication amounts to concatenation followed by
the removal of cycles (portions of the path that double back
on themselves). The use of matrices for this purpose is
already well explored (see Section 7, page 134). It is also known
to be wasteful of time and space. Here we will explore a different
technique: the use of abstract formula manipulation. We will
convert hydrocarbons into arithmetic formulae as an intermediate
stage of reduction and abstract evaluation algorithms will then map
these intermediate formulae into chemical names. We have already
seen one other application of abstract formula manipulation in
Section 4 on Electronic Circuits (see page 121), and we will see
yet another in the program simulating the processing of descriptor
formulae, this being the content of Chapter VI.

The basic idea is as follows: we take an arbitrary saturated
hydrocarbon molecule, and ignoring its hydrogen atoms, map its
carbon skeleton into a tree structure, represented as a formula.
Using this tree structure we locate a carbon atom which is on
the longest carbon chain (or on a longest carbon chain if there are
more than one) and we restructure the tree so that this carbon atom
is at the root of the tree. This restructured tree is then input to
a recursive routine which determines the length of the longest
chain ( and its corresponding Alkane name ) and determines
recursively the name of every side chain of less or equal length
( and its corresponding Alkyl name prefixed with the appropriate
integer denoting the ordinal position of the side chain on the main
chain). As an example we will use the molecule 3-methyl, 5-ethyl
octane, a two dimensional diagram of which is given at the top of
page 185. Disregarding hydrogen atoms this molecule may be
represented as:

```
  C
 C—C—C—C—C—C—C—C
  C
```

Using the topmost carbon atom as a root, this molecule maps into
the following tree structure:
If we now search for a carbon atom at the bottom most level of this tree we will find one which is at the end of the longest chain (or at the end of one of the longest chains if there is more than one). The proof of this fact is a simple exercise in graph theory and is left to the reader. The tree is now restructured using this bottom most carbon atom as its new root. The above example then maps into the following diagram:

This tree is represented as a binary formula as follows:

\[ c - ( c - ( c - ( c - ( [c - (c - (c - (c - (c - (c - (c - (c - (c - (c - (c - + (c - (c - (c)))))))))))))))) \]

which may be diagrammed as a binary tree in the following manner
The tree structure \[ \begin{array}{c}
  \text{c} \\
  \text{c} \\
  \text{c} \\
\end{array} \] is mapped onto the formula \((\text{c-} [\text{c}+\text{c}+\text{c}])\) although no example of it appears above.

**Program**

The procedures \(T1(C)\) and \(T2(L)\) below are a pair of recursive coroutines which map a molecule \(C\) onto a formula \(T1(C)\) composed, as explained above, of sums and differences of the formula atom \(\cdot c\). The procedure \(\text{CarbonList}(C)\) is used as an auxiliary and finds a list of all carbon atoms to which a given carbon is connected provided these carbon atoms have not been processed before. The mark component of a carbon atom tells whether or not it has been processed. If the mark is equal to the value of the global Boolean variable \(g\) then it has not been processed. When it has been processed the mark is set to \(\sim g\). At the termination of the routine the value of \(g\) is replaced by \(\sim g\) to reinterpret the marks for the next routine.

```plaintext
formula procedure T1(C); carbon C;
  begin integer 1; List L; mark(C) \leftarrow \sim g;
    L \leftarrow \text{CarbonList}(C); 1 \leftarrow \text{length}(L);
    T1 \leftarrow \text{if } 1=0 \text{ then } C \text{ else }
      \text{if } 1=1 \text{ then } \text{cons(BinaryFormula)}(\sim', C, T(\text{Head}(L))) \text{ else }
        \text{if } 1=2 \text{ then } \text{cons(BinaryFormula)}(\sim', C, T2(L));
  end
```
**formul procedure** T2(C) ; List L ;

T2 ← if ~ Nullist(Tail(L)) then
    cons(BinaryFormula)('+', T1(Head(L)), T2(Tail(L))) else
    T1(Head(L));

**List procedure** CarbonList(C) ; carbon C ; List L ;

begin L ← cons(Nullist) ;
for i ← 1 step 1 until 4 do
    if mark(C[3, i]) = g ∧ name(C[3, i]) = .c then
        Append(C[3, i], L) ;
CarbonList ← L ;
end ;

Having mapped the molecule C into a formula tree T1(C) we find
the last element of the longest chain in T1(C) by executing Last(T1(C)) .

The procedure Last(f) is defined as follows:

**formul procedure** Last(f) ; formula f ;

Last ← if Inst( f, a.:atom.- b.:formula ) then Last(b) else
    if Inst( f, a.:formula .+ b.:formula ) then begin
        if Length(a) > Length(b) then Last(a) else Last(b) end
    else if Inst(f, atom) then f ;

In the body of this procedure, and in the sequel, 'a .θ b' is an
abbreviation for cons(BinaryFormula)('θ', a, b) and Inst(a, b) is
a pattern matching procedure defined on pages 83 and 84 in Section 1
(on formula manipulation).
Having got the last carbon atom of the longest chain we reorganize the formula tree so that this last carbon atom occurs at its root. This is done by taking \( T_1(\text{Last}(T_1(C))) \) with \( g \) set to \( \sim g \). Finally, we take \( \text{Name}(T_1(\text{Last}(T_1(C)))) \) to get the name of the hydrocarbon.

The procedure \( \text{Name}(f) \) is defined as follows:

\[
\text{Formula Procedure} \quad \text{Name}(f) ; \quad \text{formula} \quad f ;
\]

\[
\text{Name} \leftarrow (\text{eval}(\text{SC}(1, f)) .+ \{T(\text{Length}(f))} \times \text{ane}) ;
\]

\[
\text{Formula Procedure} \quad \text{Adj}(f) ; \quad \text{formula} \quad f ;
\]

\[
\text{Adj} \leftarrow (\text{eval}(\text{SC}(1, f)) .+ \{T(\text{Length}(f))} \times \text{yl}) ;
\]

\[
\text{Formula Procedure} \quad \text{SC}(n, f) ; \quad \text{integer} \quad n ; \quad \text{formula} \quad f ;
\]

\[
\begin{align*}
\text{begin} & \quad \text{formula} \quad a, b, c ; \\
& \text{if } \text{Inst}(f, a : \text{atom} - b : \text{formula}) \text{ then } \text{SC} \leftarrow \text{SC}(n+1, b) \text{ else} \\
& \text{if } \text{Inst}(f, a : \text{atom}) \text{ then } \text{SC} \leftarrow 0 \text{ else} \\
& \text{if } \text{Inst}(f, a : \text{formula} .+ b : \text{formula} .+ c : \text{formula}) \text{ then} \\
& \quad \text{SC} \leftarrow (n - \text{Adj(Middle}(f))) .+ (n - \text{Adj(Smallest}(f)))) .+ \\
& \quad \text{SC}(n+1, \text{Largest}(f)) \text{ else} \\
& \text{if } \text{Inst}(f, a : \text{formula} .+ b : \text{formula}) \text{ then} \\
& \quad \text{SC} \leftarrow (n - \text{Adj(Smallest}(f)))) .+ \text{SC}(n+1, \text{Largest}(f)) ; \\
\text{end} ;
\end{align*}
\]

\[
\text{integer procedure} \quad \text{min}(x, y) ; \quad \text{integer} \quad x, y ;
\]

\[
\text{min} \leftarrow \text{if } x < y \text{ then } x \text{ else } y ;
\]

\[
\text{integer procedure} \quad \text{max}(x, y) ; \quad \text{integer} \quad x, y ;
\]

\[
\text{max} \leftarrow \text{if } x > y \text{ then } x \text{ else } y ;
\]
integer procedure Length(f); formula f;
    begin formula a, b;
        Length ← if Inst(f, a.:atom.- b.: formula) then 1 + Length(b) else
            if Inst(f, a.:formula .+ b.:formula ) then
                max(Length(a), Length(b)) else
                if Inst(f, atom) then 1;
    end;

Formula Procedure T(n); integer n;
    begin List names;
        names ← (.meth, .eth, . prop, .but, .pent, .hex, .hept,
            .oct, .non, . dec);
        T ← Nth(n, names);
    end;

comment : In the above procedure T(n) the abbreviation (a, b, c) has
been used for construction of the list cons(Pair)(a, cons(Pair)(b,
cons(Pair)(c, cons(Nullist)))) and the procedure Nth(n, L) has
been used. This procedure is defined on page 96;

Formula Procedure Largest(R); formula R;
    begin formula a, b, c; integer A, B, C, M;
        if Inst(R, a.:formula .+ b.:formula .+ c.:formula ) then
            begin A ← Length(a); B ← Length(b); C ← Length(c);
                M ← max( A, max(B, C) );
                Largest ← if A = M then a else if B = M then b else c;
            end;
        if Inst(R, a.:formula .+ b.:formula ) then
            begin A ← Length(a); B ← Length(b); M ← max(A, B);
                Largest ← if A = M then a else b;
            end;
    end;
Formula Procedure Smallest(R) ; formula R ;
begin formula a, b, c ; integer A, B, C, m ;
if Inst(R, a. : formula . + b. : formula . + c. : formula) then
  begin A ← Length(a) ; B ← Length(b) ; C ← Length(c) ;
  m ← min(A, min(B, C)) ;
  Smallest ← if C = m then c else if B = m then b else a ;
  end ;
if Inst(R, a. : formula . + b. : formula ) then
  begin A ← Length(a) ; B ← Length(b) ; m ← min(A, B) ;
  Smallest ← if B = m then b else a ;
  end ;
end ;

Formula Procedure Middle(R) ; formula R ;
begin formula a, b, c ; integer A, B, C ;
if Inst(R, a. : formula . + b. : formula . + c. : formula) then
  begin A ← Length(a) ; B ← Length(b) ; C ← Length(c) ;
  Middle ←
    if B=A ∧ B=C then b else
    if (A>B ∧ B>C) ∨ (C>B ∧ B>A) then b else
    if (B>A ∧ A>C) ∨ (C>A ∧ A>B) then a else
    if (A>C ∧ C>B) ∨ (B>C ∧ C>A) then c else
    if A=B then begin if A>C then b else a end else
    if A=C then begin if B>A then a else c end else
    if B=C then begin if A>B then b else a end ;
end ;
Chapter V

A Small Formal Study of the Equivalence of Evaluators and Their Data Structures

1. Introduction and Discussion

One view of evaluation is that it consists ultimately of transformations of data structures. In this chapter we restrict our attention to those evaluation processes that can be modelled by sequences of transformations of syntactic data structures. For example, the evaluation of an Algol program as performed by a compiler consists of translating text given in Algol syntax into text given in the operation code syntax of a computer, and of evaluating this translated syntax. The Algol compiler is a correct evaluator for Algol programs if the process of Algol evaluation is left invariant under the syntactic transformation or change of notation specified by the compiler.

This chapter presents, as its main result, a theorem which shows that for a certain class of models of evaluators, certain changes of syntax leave the process of evaluation invariant. This result is used to show that a model of a Lisp evaluator, based on the data structures and properties of Lisp, is faithfully imitated by a second model of Lisp evaluation based on the data structures and
properties of Formula Algol. Since these two models rely on different basic data structures, a consequence of the demonstration of faithful imitation of evaluation is that the Formula Algol data structures used in the second model to represent Lisp s-expressions behave equivalently (or are an equivalent representation for s-expressions) with respect to the set of Formula Algol representations of operations used in the imitation of the evaluation of Lisp functions. This result is our main result on the equivalence of representations. This is consistent with our view that a good way to state that data structures constitute equivalent representations is to show that they behave equivalently under given sets of equivalent operations. Thus, we believe that a meaningful discussion of equivalence of data structures should not proceed without inclusion of the equivalence of processes over them. This amounts to saying that we believe equivalence of behavior under evaluation is a better criterion than equivalence of static structure in discussing the equivalence of data representations.

The models of evaluation that we define for use in this presentation are described for theoretical purposes by simple inference systems which display the evaluation of expressions step by step in the form of a derivation (or proof) using rules
of inference which rigorously define syntactic changes. Thus, evaluation of expressions consists of transforming an input expression into a final expression by a sequence of applications of rules of syntactic transformation.

More than this, however, these models allow us to decompose the process of evaluation into two sets of primitive rules: a set of sequencing rules for defining the order of evaluation of subexpressions of an expression, and a set of equations giving properties of primitive operations on data and defining functional application. Decomposing the process of evaluation in this fashion permits us to show that the evaluation of arbitrary expressions in Lisp is preserved, provided that the sequencing rules and primitive identities of Lisp are preserved. This result allows us to verify, for example, that the Formula Algol program which imitates Lisp, given in Chapter VI, is a faithful representation of Lisp by verifying only that we have represented the several primitive identities of Lisp and the sequencing rules for Lisp evaluation.

2. An Equation for the Equivalence of Evaluators

One concept that we will need to define sharply is the concept
of what we mean by saying that two evaluators are equivalent.

A basic equation we can use to define the equivalence of evaluators is given as follows: Suppose that we represent programs with the notation \((P, d)\) in which \(P\) represents the text for a program which acts on a set of data \(d\) to transform it into a set of data \(d'\). The effect of executing \(P\) on \(d\) is determined by an evaluator \(M_1\). Thus \(d' = M_1((P, d))\). An evaluator \(M_2\) is equivalent to \(M_1\) if a translation function \(T\) can be specified from programs \(P\) and data \(d\) accepted by \(M_1\) into programs \(T[P]\) and \(T[d]\) accepted by \(M_2\) for which there exists a mapping \(T^{-1}\) defined on data of \(M_2\) into \(M_1\) and for which \(T^{-1}T = 1\) identity, such that

\[
M_1 ((P, d)) = T^{-1}[ M_2 ((T[P], T[d])) ]
\]

for all programs \(P\) and data \(d\) accepted by \(M_1\).

A valid Algol compiler is thus an example of a mapping \(T\) which translates Algol programs and Algol data into machine programs and machine data. The effect of running an Algol program is obtained, in practice, by translating Algol text and data under \(T\), interpreting the result (by processing machine instructions
interpetively in a digital computer) and by mapping the resulting machine data back into Algol data.

One objective of this chapter will be to show that the model we give of a Lisp evaluator, the second model we give in Formula Algol which imitates it, and the translation T we give from the first model to the second model together obey equation I above for the evaluation of all Lisp functions and data.

3. A Model for Evaluators

A good way to introduce our model for evaluators is to give an example accompanied by explanation. This serves to motivate the formal treatment given later.

The example we choose is a small one: an interpreter which performs addition of integers to sums not greater than 3 by evaluating arithmetic infix expressions.

First we give the syntax of infix expressions by means of a phrase structure grammar:

I. Syntax

\[
\begin{align*}
< \text{digit} > & ::= 0 | 1 | 2 | 3 \\
< \text{ae} > & ::= < \text{digit} > | ( < \text{ae} > + < \text{ae} > ) \mid < \text{expression} >
\end{align*}
\]
\(<\text{expression}\> ::= \text{eval}\ [\ <\text{ae}\ >\ ]\ |\ \text{ap}\ [\ <\text{ae}\ >\ ]\)

We then give a set of rules of inference:

II. Rules of inference

1. Rules for eval (sequencing rules giving order of evaluation)
   
   1.1 eval[(\ <\text{ae}\ >_1 + <\text{ae}\ >_2)] = ap (eval[<\text{ae}\ >_1] + eval[<\text{ae}\ >_2])
   
   1.2 eval[<\text{digit}\ >] = <\text{digit}\ >

2. Rules for ap (identities for primitive data operations)

   2.1 ap (0 + 0) = 0
   2.2 ap (0 + 1) = 1
   2.3 ap (1 + 0) = 1
   2.4 ap (1 + 1) = 2
   2.5 ap (2 + 1) = 3
   2.6 ap (1 + 2) = 3

These rules of inference are of the form \( u = v \) where \( u \) and \( v \) are phrases of the grammar of infix expressions. Informally, phrases are well-formed subexpressions derived in the grammar from non-terminal characters. A mapping \( \mu \) is specified for each rule of inference from the set of occurrences (tokens) of non-terminal characters in \( v \) into the set of occurrences of non-terminal characters in \( u \) such that \( \mu \) preserves types. I.e., if \( \alpha \) is an occurrence of a non-terminal in \( v \) and \( \mu(\alpha) \) is an occurrence of a non-terminal
in $u$ then $\mu(\alpha)$ and $\alpha$ are different occurrences of the same non-terminal character. In writing the rules of inference in the form $u = v$ if $\alpha$ and $\beta$ are the only occurrences of a given non-terminal in $u$ and $v$ respectively, then $\mu(\beta) = \alpha$, otherwise, for multiple occurrences of the same non-terminal in $u$ and $v$, the mapping $\mu$ is specified by subscripting corresponding occurrences of non-terminals in $u$ and $v$ with the same subscript.

For example, in rule 1.1
\[
\begin{align*}
u & = \text{eval}[ ( <\text{ae}>_{1} + <\text{ae}>_{2} ) ] \text{ and} \\
v & = \text{ap} ( \text{eval}[ <\text{ae}>_{1} ] + \text{eval}[ <\text{ae}>_{2} ] ) .
\end{align*}
\]
The first occurrence of the non-terminal $<\text{ae}>$ in $v$ is made to correspond to the first occurrence of the non-terminal $<\text{ae}>$ in $u$ by subscripting each with the subscript $1$. In rule 1.2, however, the unique occurrence of $<\text{digit}>$ in $v$ is made to correspond implicitly to the unique occurrence of $<\text{digit}>$ in $u$.

The mapping $\mu$ need be neither 1-1 nor onto. Thus, for example, multiple occurrences of a non-terminal $\alpha$ in $v$ may correspond to only one occurrence of $\alpha$ in $u$.

The rules of inference given by $\Pi$ may be used to transform a subset of the arithmetic infix expressions given by the grammar $I$. 
The successive steps of transformation are displayed as a trace.

III. Trace for evaluation of $(0 + (1 + 0)) + 2$

\[
\begin{align*}
\text{begin} & \quad \text{eval} \left[ (0 + (1 + 0)) + 2 \right] \\
1.1 & \quad \text{ap}( \text{eval}\left[ (0 + (1 + 0)) \right] + \text{eval}[2] ) \\
1.2 & \quad \text{ap}( \text{eval}\left[ (0 + (1 + 0)) \right] + 2 ) \\
1.1 & \quad \text{ap}( \text{ap}( \text{eval}[0] + \text{eval}[1 + 0]) ) + 2 ) \\
1.2 & \quad \text{ap}( \text{ap}( \text{eval}[0] + \text{ap}( \text{eval}[1] + \text{eval}[0]) ) + 2 ) \\
1.2 & \quad \text{ap}( \text{ap}( \text{eval}[0] + \text{ap}( 1 + 0 )) + 2 + \\
1.2 & \quad \text{ap}( \text{ap}( 0 + \text{ap}( 1 + 0 )) + 2 ) \\
2.3 & \quad \text{ap}( 0 + 1 ) + 2 \\
2.2 & \quad \text{ap}( 1 + 2 ) \\
2.6 & \quad 3
\end{align*}
\]

In this trace the initial line is labelled \textit{begin} on its left to denote the initial expression of the trace and each additional line of the trace is labelled on the left with the number of the rule of inference that applied to the previous line to transform it into the given line. A rule of inference \textit{u= v} applies to an expression \textit{e} if there is a well formed subexpression of \textit{e} (not necessarily proper) matching \textit{u}. In this match, occurrences of non-terminals in \textit{u} (if any) correspond to phrases of \textit{e}. These phrases are then substituted for corresponding occurrences of non-terminals in \textit{v} (as determined by the mapping \textit{\mu}) and the subexpression of \textit{e} matching \textit{u} is then replaced by this substituted version of \textit{v}. 
For example, the second line of the trace III arises from the first line of the trace by application of the rule of inference 1.1 since the left hand side of rule 1.1 matches a subexpression of the first line (which subexpression is, in this case, the whole expression) as follows:

\[
\text{eval } [ ( <ae>_1 + <ae>_2 ) ] \quad \text{eval } [ ( (0+(1+0)) + 2 ) ]
\]

Here the non-terminal token \(<ae>_1\) matches the subexpression \((0+(1+0))\) and the non-terminal token \(<ae>_2\) matches the subexpression 2. These subexpressions are respectively substituted for the occurrences of \(<ae>_1\) and \(<ae>_2\) in \(v = \text{ap}(\text{eval}[<ae>_1]+\text{eval}[<ae>_2])\) to yield the expression \(\text{ap}( \text{eval}[ (0+(1+0))] + \text{eval}[2])\) which replaces the subexpression matched by \(u\) in the first line of the trace (in this case the whole expression) to produce the expression in the second line of the trace. Derivation of lines of the trace continues until no further rules apply. In the case of the trace given by III, the last line consists of the expression 3, which we define to be the value of the expression \(\text{eval}[(0+(1+0))+2]\) with respect to the evaluator represented by the rules of inference.

Thus the syntax I gives a grammar for data (digits from 1 to 3).
and for expressions (arithmetic infix expressions) and the
rules of inference II give a way of defining a machine in which
evaluation of these expressions may be carried out. The application
of rules of inference to an expression may not terminate, or
may terminate with an undefined result. The above evaluator,
for example, is a partial function on its input expressions which
cannot evaluate the expression eval[(3+3)] although this expression
is well defined according to the syntax I.

In a sense, the rules of inference in the above example
decompose the act of evaluation into two sets of primitive rules
of transformation. In the case of the rules for eval, they specify
the sequence in which evaluation of infix expressions is to be carried
out. Specifically, evaluating an infix expression consists of evaluating
the left and right operands of the expression before evaluating the
expression, as in rule 1.1. The rules for ap are identities which
describe the behavior of primitive operations on data in an infix
interpreter.

This sort of decomposition of the process of evaluation will
allow us to show, under certain circumstances formalized in the
sequel, that mappings of evaluators which preserve both the 
sequencing rules and the valid identities for the primitive operations 
of the interpreter, also preserve the process of evaluation for all 
expressions. This, in effect, reduces the problem of showing that two 
evaluators of the form above behave isomorphically for the evaluation 
of expressions (P,d) to that of showing that a set of sequencing 
rules and valid primitive identities are faithfully represented in 
the other.

An example of a mapping of the arithmetic infix evaluator 
above into an evaluator for Polish prefix notation which preserves 
the evaluation of expressions can be obtained by mapping the 
syntax, the rules of inference, and the trace under a mapping T 
whose effect is given as follows:

I. Tsyntax

\[
< \text{Tdigit} > ::= 0 \mid 1 \mid 2 \mid 3 \\
< \text{Tae} > ::= < \text{Tdigit} > \mid + < \text{Tae} > < \text{Tae} > \mid < \text{Texpression} > \\
< \text{Texpression} > ::= \text{eval} < \text{Tae} > \mid \text{ap} < \text{Tae} >
\]

II. T of Rules of Inference

1. T of Rules for eval
   1.1 \text{eval} + < \text{Tae} >_1 < \text{Tae} >_2 = \text{ap} + \text{eval} < \text{Tae} >_1 \text{eval} < \text{Tae} >_2 \\
   1.2 \text{eval} < \text{Tdigit} > = < \text{Tdigit} >
2. T of Rules for ap
   2.1 ap +00 = 0
   2.2 ap +01 = 1
   2.3 ap +10 = 1
   2.4 ap +11 = 2
   2.5 ap +21 = 3
   2.6 ap +12 = 3

III. T of trace

begin   eval ++ 0 + 1 0 2
1.1   ap + eval + 0 + 1 0 eval 2
1.2   ap + eval + 0 + 1 0 2
1.1   ap + ap + eval 0 eval + 1 0 2
1.1   ap + ap + eval 0 ap + eval 1 eval 0 2
1.2   ap + ap + eval 0 ap + eval 1 0 2
1.2   ap + ap + eval 0 ap + 1 0 2
1.2   ap + ap + 0 ap + 1 0 2
2.3   ap + ap + 0 1 2
2.2   ap + 1 2
2.6   3

We observe that translating the syntax under T induces translations of both the rules of inference and the trace, and that the translated trace is a valid derivation under the translated rules of inference.
That is, T is an example of a transformation that preserves traces, or equivalently, preserves evaluation. We will later present a set
of sufficient properties on \( T \) in order that \( T \) be trace (or evaluation) preserving and we will prove that the trace is preserved.

Another way of stating that \( T \) is evaluation preserving is to state that the process of evaluation is invariant under the change of notation specified by \( T \). The theorem we present next showing properties of a class of transformations \( T \) which preserve traces can thus be interpreted as a theorem specifying a class of changes of notation under which a given abstract process of evaluation is left invariant.

4. Formal Treatment

We proceed in this section to give a formal treatment of a set of conditions on \( T \) which guarantees that \( T \) preserves traces.

Conventional Definitions for Phrase Structure Grammars

One set of definitions for phrase structure grammars is given by Floyd [23]. Essentially this is done as follows. Characters are represented by Latin capitals or by subscripted Latin capitals, strings are represented by lower case Latin letters or by subscripted lower
case Latin letters, and the null string is denoted $\Lambda$, and is an identity under concatenation of strings. If $x$ and $y$ are strings, $xy$ denotes the concatenation of $x$ and $y$. A production $U \rightarrow u$ is an ordered pair consisting of a character $U$ and a non-empty string $u$. The relation $w \rightarrow v$ holds with respect to a set of productions $P$, if there are strings $x$ and $y$ and a character $U$ such that $w = xu y$ and $v = xuy$ and $U \rightarrow u$ is a production of $P$.

The relation $w \Rightarrow v$ holds with respect to a set of productions $P$ if there is a finite sequence $w = w_0, w_1, \ldots, w_n = v \ (n \geq 0)$ such that $w_{i-1} \rightarrow w_i \ (1 \leq i \leq n)$. A sequence $w = w_0, w_1, \ldots, w_n = v$ for which $w_{i-1} \rightarrow w_i \ (1 \leq i \leq n)$ with respect to a set of productions $P$ is said to be a derivation of $v$ from $w$ and $v$ is a $w$-derivative.

A phrase structure grammar $G$ is a set of productions containing exactly one character $S$ which appears only on the left of ' $\rightarrow$ ' and a non-empty set of characters (called terminal characters and denoted $TC(G)$) which appear only on the right of ' $\rightarrow$ '. The non-terminal characters of $G$ (denoted $NT(G)$) are those which appear on the left of ' $\rightarrow$ ' in the productions of $G$. The sentential forms of $G$, denoted $SF(G)$, are the derivatives of $S$ with respect to the productions of $G$. The sentences of $G$ are those sentential forms of $G$ composed only of terminal characters. This set of sentences is denoted $L(G)$ and $L(G)$ is also called the phrase structure language.
associated with the phrase structure grammar $G$. The phrases of $G$, denoted $\text{Phrases}(G)$, are the set of derivatives of non-terminal characters of $G$ with respect to the productions of $G$.

**Graph Theoretical Formulation of Phrase Structure Grammars**

We shall find it convenient in our proofs to use phrase markers (or equivalently parses or syntax trees) of sentences in formulating proofs of our theorems. This seems to be the proper representation. The following set of definitions is based on a treatment of phrase markers given by Floyd in his NATO Lecture Series [23]. The use of phrase markers in discussing phrase structure grammars can be shown to be equivalent to the use of strings and derivations.

A directed graph $G$ is a set of nodes or vertices $V$ together with a set $E$ of (directed) edges consisting of ordered pairs of vertices of $V$ ($E \subseteq V \times V$). A path from vertex $a$ to vertex $b$ in $G$ is a sequence of edges of $G$ of the form $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ such that $y_i = x_{i+1}$ ($1 \leq i \leq n-1$) and such that $a = x_1$ and $b = y_n$. The length of a path is the number of
edges in it. Given a graph G, we can define a relation

Path : V×V → (true, false) such that Path(a, b) is true

if and only if there exists a path from a to b in G. It follows
from the definition of a path that this relation is transitive,
i.e. Path(a, b) ∧ Path(b, c) → Path(a, c). If, in the graph G,
a relation on its vertices is anti-symmetric ( R(a, b) → ~R(b, a) )
as well as transitive, we will call it a descendence relation. A
descendence relation will be signified by the notation a > b,
and we will say that a dominates b, that a is an ancestor of b,
or that b is a descendant of a.

A tree is a directed graph G such that (1) for any two
vertices in G there is at most one path between them, and (2)
there exists a vertex r in G, called the root of G, such that for
any vertex x in G distinct from r, there is a path from r to x.
Some consequences of this definition are: (1) The root of a tree
is unique. [Proof: Suppose there existed a tree with two distinct roots
r₁ and r₂. Applying property (2) twice there exist paths from r₁ to
r₂ and from r₂ to r₁ since each is a root and is distinct from the other.
But this contradicts property (1) since we have exhibited two vertices
with more than one path between them.], (2) The relation

† Here, and in the sequel, '→' is a sign denoting logical implication.
Path(a, b) on a tree is a descentence relation. [Proof: Path(a, b) is transitive as a consequence of its definition. In addition, in a tree, it is anti-symmetric as well, since if Path(a, b) is true then there is a path from vertex a to vertex b, and Path(b, a) cannot be true without violating property (1).]

A leaf of a tree G is a vertex of G having no other distinct vertices as descendants. We will denote the property of being a leaf of a tree by the notation Leaf(l, G). This may be defined as Leaf(l, G) = ~ [ ∃ x (x ∈ G) ∧ Path(l, x) ].

In the trees used to represent grammatical derivations it is necessary to specify an order among the nodes. This may be done by associating Dewey Decimals with the nodes. A Dewey Decimal will be defined as a list of positive integers: [n_1, n_2, ..., n_k]. (A more familiar representation is obtained by replacing commas by periods and by removing square brackets to give n_1·n_2·...·n_k).

Representing nodes by points and edges by arrows, an example of a tree with associated Dewey Decimals is:

```
  [1]
 /  |  \
[1, 1] [1, 2] [1, 3]
      |    |
   [1, 2, 1] [1, 2, 2]
```
The assignment of Dewey Decimals to a tree \( G \) has the following properties:

1. The Dewey Decimal \([1]\) is assigned to the root of \( G \).

2. If \([n_1, n_2, \ldots, n_k]\) is a Dewey Decimal associated with a vertex \( v \) of \( G \) and if \( n_k > 1 \) then there exists a vertex \( x \) of \( G \) distinct from \( v \) whose Dewey Decimal is \([n_1, n_2, \ldots, n_{k-1}, n_{k-1}]\).

3. If \([n_1, n_2, \ldots, n_k]\) is a Dewey Decimal associated with a vertex \( v \) of \( G \) and if \( k > 1 \) then there exists a vertex \( x \) of \( G \) distinct from \( v \) whose Dewey Decimal is \([n_1, n_2, \ldots, n_{k-1}]\).

A tree with Dewey Decimals associated with its nodes in accord with these properties will be called an \textit{ordered tree}. In an ordered tree, both ancestor-descendant and left-right relations may be defined as follows:†

1. Vertex \( a \) is an ancestor of vertex \( b \) (equivalently \( b \) is a descendant of \( a \)) if
   i. \( \text{length}(\text{DeweyDecimal}(a)) < \text{length}(\text{DeweyDecimal}(b)) \).
   ii. \( \text{DeweyDecimal}(a)[i] = \text{DeweyDecimal}(b)[i] \) is \textit{true} for \( (1 \leq i \leq \text{length}(\text{DeweyDecimal}(a))) \).

2. Vertex \( a \) is left of vertex \( b \) (equivalently \( b \) is right of \( a \)) if there exists a \( k \) such that
   \( 0 < k \leq \min(\text{length}(\text{DeweyDecimal}(a)), \text{length}(\text{DeweyDecimal}(b))) \)
   and for \( i \leftarrow 1 \) step 1 until \( k-1 \) \( \text{DeweyDecimal}(a)[i] = \text{DeweyDecimal}(b)[i] \) and \( \text{DeweyDecimal}(a)[k] < \text{DeweyDecimal}(b)[k] \).

A father-son relation, \( \text{father}(a, b) \iff \text{son}(b, a) \), may be defined with father equivalent to immediate ancestor and son equivalent to

† In these definitions, \( \text{DeweyDecimal}(x) \) is a list whose length function is defined on page 97, and given that \( L \) is a list \( L[i] \) is a notation for the \( i \)th object in \( L \).
immediate descendant by replacing condition (1) i above with
the condition (1) i'. \( \text{length(DeweyDecimal(b))} - \text{length(DeweyDecimal(a))} = 1 \).

A subtree of an ordered tree may be converted to an ordered tree
by replacing the maximum common initial sequence of indices in
each of its DeweyDecimals with 1 and by detaching it from the
super-tree. The reverse of this process gives a way of composing
two ordered trees \( t_1 \) and \( t_2 \) into a new ordered tree with \( t_2 \) a
subtree of \( t_1 \). Namely, we choose a leaf \( 1 \) of \( t_1 \) equal to the root
of \( t_2 \) and replace the initial 1 in each Dewey Decimal of \( t_2 \) with
the Dewey Decimal of 1.

\[
\text{Subst}(\ell, t_1, t_2)
\]

The ordered tree produced by this operation is called the ordered
tree derived by substituting \( t_2 \) for 1 in \( t_1 \). This derived tree is
denoted \( \text{Subst}(1, t_1, t_2) \).
Given any node \( x \) in a tree \( t \), a unique subtree is determined by considering all nodes \( y \) in \( t \) such that \( y = x \) or \( y \) is a descendant of \( x \). This subtree is denoted \( D(x, t) \), the subtree dominated by \( x \) in \( t \). If \( x \) is a subtree of the ordered tree \( t \), then we define \( D(x, t) = D(\text{Root}(x), t) \).

**Phrase Markers**

Given a phrase structure grammar \( g \), let \( p \) be a finite ordered tree whose vertices are labelled with members of the vocabulary of \( g \). \( p \) is a **phrase marker** of a sentential form, or phrase of \( g \), if whenever \( v \) is a vertex labelled with a non-terminal character \( U \) of \( g \), and the sons of \( v \) in left to right order are labelled \( u_1, u_2, \ldots, u_k \), then \( U \rightarrow u_1 u_2 \ldots u_k \) is a production of \( g \).

Let \( p \) be a phrase marker. Then \( \text{String}(p) \) is the phrase or sentential form of \( g \) obtained by concatenating the labels of the leaves of \( p \) in left to right order. Let \( r = \text{Root}(p) \) and \( s = \text{String}(p) \).

It is easy to see that a phrase marker \( p \) is in one-to-one correspondence with a set of derivations of \( s \) from \( r \), since each subtree of \( p \) determined by a vertex labelled with a non-terminal, and by its immediate sons corresponds to the application of a production in \( g \) to a character in an \( r \)-derivative.
Phrase markers tell explicitly which characters are operated upon and by which production to produce sentential forms (in contrast to derivations, which don't). Phrase markers also make it easy to give a definition for ambiguity. An ambiguous phrase structure grammar is one in which there exists a sentential form \( s \) and two distinct phrase markers \( p_1 \) and \( p_2 \) such that
\[
\text{String}(p_1) = \text{String}(p_2).
\]
An unambiguous phrase structure grammar is one in which one and only one phrase marker corresponds to each distinct sentential form.

Given a phrase structure grammar \( g \), we will construct, for each of its productions \( U \rightarrow u_1 u_2 \ldots u_k \) a phrase marker \( p \) in which \( U \) is the label of the root of \( p \), which root is the father of \( k \) immediate sons, one for each \( u_i \) (\( 1 \leq i \leq k \)), and in which if \( u_i \) is left of \( u_j \) in the string \( u_1 u_2 \ldots u_k \), then the son labelled \( u_i \) is left of the son labelled \( u_j \) in \( p \).

Let \( w \) and \( v \) be sentential forms of \( g \). If \( w \rightarrow v \) with respect to a production \( U \rightarrow u \) in \( g \) then there exist strings \( p \) and \( q \) such that \( w = pUq \) and \( v = puq \). Let \( \& \) denote the phrase marker such that \( x = \text{String}(\&) \). Then we say \( \& \rightarrow \tilde{y} \) if there is a phrase marker \( \tilde{y} \) corresponding to a production \( U \rightarrow u \) and a leaf of \( \& \) labelled \( U \).
such that \( \mathcal{V} = \text{Subst}(U, \mathcal{F}, \mathcal{P}) \). The relation \( \hat{w} \xrightarrow{\star} \mathcal{V} \) holds with respect to a set of productions \( P \) (in phrase marker form) if there is a finite sequence of phrase markers \( \hat{w} = \hat{w}_0, \hat{w}_1, \hat{w}_2, \ldots, \hat{w}_n = \mathcal{V} \) such that \( \hat{w}_{i-1} \xrightarrow{\star} \hat{w}_i \) (\( 1 \leq i \leq n \)). We note that \( \hat{w}_i \) is a subgraph of \( \hat{w}_j \) for \( i \leq j \) in this sequence.

Thus we see that the use of productions in phrase marker form can be made to correspond isomorphically to the use of productions in string form for deriving sentential forms, the isomorphism being given by \( w \xrightarrow{\star} v \) if and only if \( \hat{w} \xrightarrow{\star} \mathcal{V} \) where \( w = \text{String}(\hat{w}) \) and \( v = \text{String}(\mathcal{V}) \). Given that the use of phrase markers and strings is equivalent in the treatment of phrase structure grammars we will use phrase markers from now on to discuss derivations of sentential forms in phrase structure grammars.

Rules of Inference, Inference Systems and Traces

We may now define rules of inference and state what we mean by applying a rule of inference to a string. Let \( \mathcal{A} \) be a phrase marker corresponding to the string \( a \) in the unambiguous phrase structure grammar \( g \), i.e. \( a = \text{String}(\mathcal{A}) \). A rule of inference \( u = v \) is an ordered pair of phrases of \( g \) such that there exist
phrase markers $\hat{u}, \hat{v}$ of $u$ and $v$ respectively for which $\text{Root}(\hat{u}) = \text{Root}(\hat{v})$ and such that there exists a mapping $\mu$ of the set of occurrences of non-terminals in $v$ into the set of occurrences of non-terminals in $u$ such that for any occurrence $x$ of a non-terminal in $v$, $x$ and $\mu(x)$ are occurrences of the same non-terminal.

We say that a sentence $t$ of $g$ is a derivative of a sentence $s$ of $g$ with respect to the rule of inference $u=v$, (written $s\rightarrow t \text{ wrt. } u=v$) provided

1. There exists $\hat{M}$ such that $\text{Leaf}(\beta, \hat{M}) \wedge \beta = \text{Root}(\hat{u}) = \text{Root}(\hat{v})$
   $\wedge \hat{s} = \text{Subst}(\beta, \hat{M}, D(\hat{u}, \hat{s})) \wedge$
   $\hat{f} = \text{Subst}(\beta, \hat{M}, D(\hat{v}, \hat{f}))$ and

2. $\forall \alpha \quad \text{NT}(\alpha) \wedge \text{Leaf}(\alpha, \hat{v}) \Rightarrow$
   $\text{Leaf}(\mu(\alpha), \hat{u}) \wedge D(\alpha, \hat{f}) = D(\mu(\alpha), \hat{s})$

An inference system is an ordered pair $(g, R)$ consisting of a phrase structure grammar $g$ and a set of rules of inference $R$.

A trace in an inference system $(g, R)$ is a sequence $s_0, s_1, \ldots, s_k$ of sentences of $g$ such that $s_{i-1} \rightarrow s_i$ ($1 \leq i \leq k$) with respect to a rule of inference in $R$.

Theorem on Invariance of Evaluation under Change of Notation

Let $g$ and $g'$ be unambiguous phrase structure grammars.
Let T be a mapping from g into g' with the following properties:

(1) T is a 1-1 onto mapping of NT(g) onto NT(g') .

(2) If U → w is a production of g then T(U) → T(w)
    is a production of g' such that the image under T of U
    agrees with the mapping in (1) , and such that T is a
    1-1 onto mapping of the set of occurrences of non-
    terminals in w onto the set of occurrences of non-
    terminals in T(w) in such a way that if α is an
    occurrence of a non-terminal in w then the image of
    α in T(w) under T is an occurrence of T(α).

Given that T is defined from phrase structure grammar g onto
grammar g' it induces a mapping on the rules of inference and
rules of inference and traces of inference systems of the form (g, R) onto inference
systems of the form (T(g), T(R)). We will now demonstrate that
if T is defined, as above, from g → g' = T(g) it is also defined
from phrase markers, strings, rules of inference and traces of
any system of inference (g, R) into corresponding phrase markers,
strings, rules of inference and traces of (T(g), T(R)) in such a
way that derivations of traces in (g, R) are preserved under T.

Given a leaf l of a tree t, there is a unique path from
Root(t) to l. Consider the set of all paths from Root(t) to the leaves
of t. The depth of t is the maximum of the lengths of the paths in this set.
Given that T is defined from productions of g into productions of g', T is defined on phrase markers inductively and T has certain properties. The proof proceeds by induction on the depth of phrase markers.

(1) If t is a phrase marker of depth 1, then, by the definition of phrase marker, if \( u_1, u_2, \ldots, u_k \) are the sons of Root(t) in left to right order, then \( \text{Root}(t) \rightarrow u_1u_2\ldots u_k \) is a production of g.

Since T is defined for productions, we define T(t) to be the phrase marker corresponding to \( T(\text{Root}(t) \rightarrow u_1u_2\ldots u_k) \). Namely, we set (i) \( \text{Root}(T(t)) = T(\text{Root}(t)) \) and set the sons of Root(T(t)) equal to the characters of \( T(u_1u_2\ldots u_k) \) in left to right order. We know by the definition of T on productions that T induces a 1-1 mapping of occurrences of non-terminals in \( u_1u_2\ldots u_k \) onto the set of occurrences of their T images. Specifically, this means that

(ii) \( \text{NT}(l) \wedge \text{Leaf}(l, t) \Rightarrow \text{NT}(T(l)) \wedge \text{Leaf}(T(l), T(t)) \) and (iii) there exists a 1-1 onto map M of the non-terminal leaves l of t onto the non-terminal leaves of T(t) which agrees with \( T: \text{NT}(g) \rightarrow \text{NT}(g') \).

(iv) We also note that \( T(D(a, t)) = D(T(a), T(t)) \) for all non-terminal nodes a of t since: Case I - if \( a = \text{Root}(t) \) then we note that \( T(t) = T(D(a, t)) \), \( \text{Root}(T(t)) = T(\text{Root}(t)) = T(a) \), and \( T(t) = D(\text{Root}(T(t)), T(t)) = D(T(a), T(t)) \). Hence, \( T(D(a, t)) = D(T(a), T(t)) \); Case II - if \( \text{Leaf}(a, t) \) then \( D(a, t) = a \).
and $T(D(a, t)) = T(a)$. But $Leaf(T(a), T(t))$ by ii, so $D(T(a), T(t)) = T(a)$. Therefore, $T(D(a, t)) = T(a) = D(T(a), T(t))$. (v) If $a$ is any non-terminal node of $t$ then $T(a)$ is a non-terminal node in $T(t)$. I.e.

$NT(a) \land a \in t \Rightarrow NT(T(a)) \land T(a) \in T(t)$. Case I - Suppose $a = Root(t)$. Then $NT(a)$ and $T(a) = T(Root(t)) = Root(T(t))$, but $Root(T(t))$ is non-terminal and belongs to $T(t)$ by construction since it is the left hand side of a production of $T(g)$. Case II - If $Leaf(a, t)$ and $NT(a)$ then iii shows that $NT(T(a))$ and $T(a) \in T(t)$ since $Leaf(T(a), T(t))$.

(2) Suppose $T$ is defined on all phrase markers $y$ of depth $\leq k$ in such a fashion that (i) $Root(T(y)) = T(Root(y))$, (ii) $NT(l) \land Leaf(l, y) \Rightarrow NT(T(l)) \land Leaf(T(l), T(y))$, (iii) there exists a 1-1 onto map $M$ of non-terminal leaves of $y$ onto non-terminal leaves of $T(y)$, (iv) for all non-terminal nodes $a$ in $y$, $T(D(a, y)) = D(T(a), T(y))$, and (v) $NT(a) \land a \in y \Rightarrow NT(T(a)) \land T(a) \in T(y)$ for all nodes $a$ of $t$.

Then consider a phrase marker $t$ of depth $k+1$. The subtree $x$ of $t$ whose root is $Root(t)$ and whose leaves are the sons of $Root(t)$ is a phrase marker of depth 1. By (1) above $T$ is defined on $x$ and properties (i), (ii), (iii), (iv) and (v) hold for $T(x)$. Consider also the subtrees whose roots are $l_1, l_2, \ldots, l_k$ the non-terminal leaves of $x$. These subtrees are denoted $D(l_i, t)$ for $1 \leq i \leq k$. Since these subtrees
are phrase markers of depth \( \leq k \), \( T \) is defined for them and properties (i), (ii), (iii), (iv) and (v) hold. We further note that \( t \) is the result of a sequence of substitutions of each \( D(l_i, t) \) for each \( l_i \) in \( x \) (\( 1 \leq i \leq k \)). This sequence of substitutions may be expressed \( a_0 = x, \ a_i = \text{Subst} (l_i, a_{i-1}, D(l_i, t)) \) for \( 1 \leq i \leq k \), \( t = a_k \). We know by (ii) that \( T(l_i) \) is a leaf of \( T(x) \) (\( 1 \leq i \leq k \)) and by (i) that \( \text{Root} (T(D(l_i, t))) = T(\text{Root}(D(l_i, t))) = T(l_i) \) (\( 1 \leq i \leq k \)). Hence, a sequence of substitutions is defined

\[
T(a_0) = T(x), \ T(a_i) = \text{Subst}(T(l_i), T(a_{i-1}), T(D(l_i, t))), \ T(t) = T(a_k).
\]

Hence, \( T(t) \) is well defined and we can note the following properties:

(i) by construction \( \text{Root}(T(t)) = \text{Root}(T(x)) = T(\text{Root}(x)) = T(\text{Root}(t)) \),

hence, \( \text{Root}(T(t)) = T(\text{Root}(t)) \),

(ii) the non-terminal leaves of \( t \)

are a disjoint union \( A \) of the non-terminal leaves of \( D(l_i, t) \) (\( 1 \leq i \leq k \)).

By construction, the non-terminal leaves of \( T(t) \) are a disjoint union \( T(A) \) of the non-terminal leaves of \( T(D(l_i, t)) \) (\( 1 \leq i \leq k \)). But since property (ii) holds for each of the non-terminal leaves in \( A \) by virtue of the inductive hypothesis, we see that \( \text{NT}(l) \wedge \text{Leaf}(l, t) \Rightarrow \text{NT}(T(l)) \wedge \text{Leaf}(T(l), T(t)) \) for all non-terminal leaves of \( t \),

(iii) by the inductive hypotheses, 1-1 onto mappings \( M \) exist from the non-terminal leaves of \( D(l_i, t) \) (\( 1 \leq i \leq k \)) onto the non-terminal leaves of \( T(D(l_i, t)) \) (\( 1 \leq i \leq k \)), but the non-terminal leaves of \( t \) and \( T(t) \) respectively are disjoint unions of the non-terminals of \( D(l_i, t) \) and \( T(D(l_i, t)) \) (\( 1 \leq i \leq k \)).
respectively. By a composition of M's defined on the non-terminal leaves of $D(l_i, t)$ \(1 \leq i \leq k\) we can define a 1-1 onto mapping $M$ on the disjoint union, so that $M$ extends to map all non-terminal leaves of $t$ onto those of $T(t)$ and is 1-1 onto, (iv) we know from the inductive hypotheses that for each node $z$ in $D(l_i, t)$, $T(D(z, D(l_i, t))) = D(T(z), T(D(l_i, t)))$ \(1 \leq i \leq k\). Consider $t$. Choose an arbitrary non-terminal node $a$ in $t$. Then $a = \text{Root}(t)$ or $a \in D(l_j, t)$ for some $j$, $1 \leq j \leq k$. If $a = \text{Root}(t)$ then $D(a, t) = t$ so $T(D(a, t)) = T(t)$ and $T(a) = T(\text{Root}(t)) = \text{Root}(T(t))$. But $T(t) = D(\text{Root}(T(t)), T(t)) = D(T(a), T(t))$. Therefore $T(D(a, t)) = D(T(a), T(t))$. Now consider $a \in D(l_j, t)$. Then $T(D(a, t)) = T(D(a, D(l_j, t)))$. By the inductive hypotheses $T(D(a, D(l_j, t))) = D(T(a), T(D(l_j, t)))$ and by construction $D(T(a), T(D(l_j, t))) = D(T(a), T(t))$. Hence, $T(D(a, t)) = D(T(a), T(t))$ for all non-terminal nodes $a$ in $t$, (v) by the inductive hypotheses $NT(a) \land a \in D(l_i, t) \Rightarrow NT(T(a)) \land T(a) \in T(D(l_i, t)) \land T(a) \in T(t)$. If $a \in t$ then $a = \text{Root}(t)$ or $a \in D(l_j, t)$ for some $j$, $1 \leq j \leq k$. If $a = \text{Root}(t)$ then $T(a) = T(\text{Root}(t)) = \text{Root}(T(t))$ so $NT(T(a)) \land T(a) \in T(t)$. If $a \in D(l_j, t)$ for some $j$ then $T(D(l_j, t))$ is a subtree of $T(t)$ by construction, and $NT(T(a)) \land T(a) \in T(D(l_j, t)) \Rightarrow NT(T(a)) \land T(a) \in T(t)$ for all $a \in t$.

By induction, then, $T$ is well defined for phrase markers of all depths and $T$ has the following properties:
i. \( T(\text{Root}(t)) = \text{Root}(T(t)) \)

ii. \( \text{NT}(l) \land \text{Leaf}(l, t) \Rightarrow \text{NT}(T(l)) \land \text{Leaf}(T(l), T(t)) \)

iii. There is a 1-1 onto mapping \( M \) from non-terminal leaves of \( t \) onto non-terminal leaves of \( T(t) \)

iv. For any non-terminal node \( a \) in \( t \), \( T(D(a, t)) = D(T(a), T(t)) \)

v. \( \text{NT}(a) \land a \in t \Rightarrow \text{NT}(T(a)) \land T(a) \in T(t) \)

vi. \( t = y \Rightarrow T(t) = T(y) \) since \( T \) is well defined on phrase markers.

Provided that \( \alpha = \text{Root}(m) \) and \( \text{Leaf}(\alpha, t) \) it is straightforward to show by induction on the length of the path from \( \text{Root}(t) \) to \( \text{Root}(m) \) in \( t \) that

vii. \( T\left( \text{Subst}(\alpha, t, m) \right) = \text{Subst}(T(\alpha), T(t), T(m)) \).

We note that this property is consistent with the substitution operations used in the definition of \( T \) since replacing \( a_i \) in

\[
T(a_i) = \text{Subst}\left( T(1_i), T(a_{i-1}), T(D(1_i, t)) \right)
\]

with

\[
a_i = \text{Subst}\left( l_i, a_{i-1}, D(1_i, t) \right)
\]

yields

\[
T(\text{Subst}(l_i, a_{i-1}, D(1_i, t))) = \text{Subst}(T(l_i), T(a_{i-1}), T(D(1_i, t)))
\]

[see equation II above, page 221].

Our next observation is that since \( T \) is 1-1 onto of finite sets in all of the above definitions, there exists a 1-1 onto function \( T^{-1} \) defined on \( \text{NT}(g') \rightarrow \text{NT}(g) \), on productions of \( g' \) onto productions
of $g$ and on phrase markers of $g'$ onto phrase markers of $g$. This inverse mapping has the property that

$$viii. \quad T T^{-1} = T^{-1} T = \text{identity}$$

for all classes of arguments. The proofs that $T^{-1}$ has the same properties as $T$ now proceeds by symmetry using the proofs for $T$.

Since $T$ is defined for phrase markers it induces a mapping on rules of inference. If $u = v$ is a rule of inference then there exist phrase markers $\hat{u}$ and $\hat{v}$ such that $\text{Root}(\hat{u}) = \text{Root}(\hat{v})$, $u = \text{String}(\hat{u})$ and $v = \text{String}(\hat{v})$ and such that there exists a mapping $\mu$ from occurrences of non-terminals in $v$ into occurrences of non-terminals in $u$. We define $T(u = v)$ to be $T(u) = T(v)$.

By property (i) $\text{Root}(u) = \text{Root}(v) \Rightarrow T(\text{Root}(u)) = T(\text{Root}(v))$ $\Rightarrow \text{Root}(T(u)) = \text{Root}(T(v))$. Furthermore $\bar{\mu} = T(\mu)$ can be defined by setting $\bar{\mu} = T\mu T^{-1}$ since by property (iii) there is a 1-1 mapping induced by $T^{-1}$ on non-terminal leaves of $T(\hat{v})$ onto non-terminal leaves of $\hat{v}$, whence non-terminal leaves of $\hat{v}$ are mapped by $\mu$ into non-terminal leaves of $\hat{u}$, and then mapped by $T$ onto leaves of $T(\hat{u})$. This induces a mapping $\bar{\mu}$ from occurrences of non-terminals in $T(\hat{v})$ into occurrences of non-terminals in $T(u)$. Thus $T(u = v)$ is well defined.
We write the definition of $\bar{\mu}$ as rule (ix):

\[ \text{xix. } \bar{\mu} = T\mu T^{-1} \text{, by definition.} \]

The rules (i) through (ix) are a collection of definitions, properties and lemmas that we will need to prove the final theorem that $T$ is trace preserving.

$T$ is defined from sentences of $g$ onto sentences of $g'$ as follows: If $u$ is a sentence of $g$, let $\overline{u}$ be the unique phrase marker in $g$ such that $u = \text{String}(\overline{u})$. The existence and uniqueness of $\overline{u}$ is guaranteed by the non-ambiguity of $g$. Set $T(u) = \text{String}(T(\overline{u}))$.

Once $T$ is defined on sentences of $g$ it is defined on the sentences of traces of any inference system $(g, R)$. Let $w_0, w_1, \ldots, w_n$ be a trace in $(g, R)$. Then $T(w_0), T(w_1), \ldots, T(w_n)$ is a sequence of sentences in $(T(g), T(R))$. To show that $T(w_0), T(w_1), \ldots, T(w_n)$ is a trace of $(T(g), T(R))$, it is sufficient to show that $T(w_{i-1}) \rightarrow T(w_i)$ ($1 \leq i \leq n$) with respect to some rule of inference chosen from $T(R)$. This will be the case if whenever $s \rightarrow t$ with respect to a rule of inference $u = v$ in $(g, R)$ then $T(s) \rightarrow T(t)$ with respect to $T(\overline{u} = \overline{v})$ in $(T(g), T(R))$.

We introduce one final notational convention to make the proof that $T$ is trace preserving more readable. For any object $x$, let
be a synonym for $T(x)$. The properties, definitions and lemmas needed to prove that $T$ is trace preserving are summarized as follows:

i. $T(\text{Root}(t)) = \text{Root}(T(t))$, and $T^{-1}(\text{Root}(t)) = \text{Root}(T^{-1}(t))$ for all phrase markers $t$.

ii. $\text{NT}(l) \wedge \text{Leaf}(l,t) \Rightarrow \text{NT}(T(l)) \wedge \text{Leaf}(T(l),T(t))$ and $\text{NT}(l) \wedge \text{Leaf}(l,t) \Rightarrow \text{NT}(T^{-1}(l)) \wedge \text{Leaf}(T^{-1}(l),T^{-1}(t))$

iii. There is a 1-1 mapping $M$ of non-terminal leaves of $t$ onto $T(t)$ and from $t$ onto $T^{-1}(t)$

iv. For any non-terminal node $a \in t$, $T(D(a,t)) = D(T(a),T(t))$ and $T^{-1}(D(a,t)) = D(T^{-1}(a),T^{-1}(t))$

v. $\text{NT}(a) \Rightarrow \text{NT}(T(a))$ and $\text{NT}(a) \Rightarrow \text{NT}(T^{-1}(a))$ for any $a \in t$

vi. For $t$ and $y$ phrase markers $t = y \Rightarrow T(t) = T(y)$

vii. $T(\text{Subst}(a,x,y)) = \text{Subst}(T(a),T(x),T(y))$

viii. $TT^{-1} = T^{-1}T = \text{identity}$

ix. $\mu = T\mu T^{-1}$

x. $\hat{u} = T(\hat{u}), \hat{v} = T(\hat{v}), \hat{\phi} = T^{-1}(\hat{\phi})$ and $\hat{u} = T^{-1}(\hat{u})$ by notational convention and by viii.

Proof that $T$ is trace preserving:

If $s \rightarrow t \text{ wrt. } u = v$ then, by definition, we know that

1. there exists $\hat{M}$ such that Leaf($\beta, \hat{M}$) $\wedge \beta = \text{Root}(\hat{d}) = \text{Root}(\hat{\phi})$ $\wedge$

   $\hat{\phi} = \text{Subst}(\beta, \hat{M}, \text{D}(\hat{u}, \hat{\phi})) \wedge \hat{\tau} = \text{Subst}(\beta, \hat{M}, \text{D}(\phi, \hat{\tau}))$, and

2. $\forall \alpha \text{ NT}(\alpha) \wedge \text{Leaf}(\alpha, \hat{\phi}) \Rightarrow \text{Leaf}(\mu(\alpha), \hat{\alpha}) \wedge \text{D}(\alpha, \hat{\tau}) = \text{D}(\mu(\alpha), \hat{\phi})$, where $\mu$ and $\beta$ are associated with the definition of the rule of inference $u = v$. To show that $\bar{s} \rightarrow \bar{t} \text{ wrt. } \bar{u} = \bar{v}$,
(1) Choose $\hat{M} = T(\hat{M})$

(a) by ii, $\text{Leaf}(\beta, \hat{M}) \Rightarrow \text{Leaf}(\hat{\beta}, \hat{M})$,

(b) by i, $\beta = \text{Root}(\hat{u}) = \text{Root}(\hat{v}) \Rightarrow \hat{\beta} = \text{Root}(\hat{u}) = \text{Root}(\hat{v})$,

(c) by vii, iv, $s = \text{Subst}(\beta, \hat{M}, D(\hat{u}, s)) \Rightarrow \hat{s} = \text{Subst}(\hat{\beta}, \hat{M}, D(\hat{u}, \hat{s}))$ and

\[
\hat{t} = \text{Subst}(\beta, \hat{M}, D(\hat{v}, \hat{t})) \Rightarrow \hat{t} = \text{Subst}(\hat{\beta}, \hat{M}, D(\hat{v}, \hat{t})).
\]

(2) $\forall \alpha \ NT(\alpha) \land \text{Leaf}(\hat{\alpha}, \hat{v}) \Rightarrow (\text{by } v, ii, viii)$

\[
\begin{align*}
NT(\ T^{-1}(\alpha) \ ) \land \text{Leaf} \ (\ T^{-1}(\alpha), \ \hat{v}) & \Rightarrow (\text{by } s \rightarrow t \ \text{wrt. } u=v, (2)) \\
\text{Leaf}(\mu(\ T^{-1}(\alpha) \ ), \ \hat{u}) \land D(\ T^{-1}(\alpha), \ \hat{t}) & = D(\mu(\ T^{-1}(\alpha)), \ s) \\
& \Rightarrow (\text{by } ii, viii, vi, iv) \text{Leaf}(\ T \mu T^{-1}(\hat{\alpha}), \hat{u} ) \land D(TT^{-1}(\alpha), \hat{t}) = D(\ T \mu T^{-1}(\hat{\alpha}), \ s) \\
& \Rightarrow (\text{by } vii, ix) \\
\text{Leaf}(\ m(\hat{\alpha}), \hat{u}) \land D(\hat{\alpha}, \hat{t}) & = D(\ m(\hat{\alpha}), \hat{s} ).
\end{align*}
\]

Thus we see that the properties required to show that $T(s) \rightarrow T(t)$ with respect to $T(u) = T(v)$ have been proven and therefore the translations of traces under $T$ are traces under translated rules of inference. This concludes the formal demonstration.

5. Imitating Lisp with Formula Algol

The following section exhibits a model of a Lisp evaluator in the form of a syntactic inference system, and it exhibits a second model which is a translation of this evaluator whose data
are Formula Algol formulae in which evaluation of Lisp expressions is preserved.

I. Syntax for Lisp Evaluator which is Closed under Rules of Inference

\[
\begin{align*}
<\text{form}> & ::= <\text{constant}> | <\text{variable}> | <\text{function}>[<\text{argument list}>] | \\
& \quad | <\text{conditional expression}> | \text{ap}[<\text{function}>; [<\text{argument list}>]] \\
<\text{constant}> & ::= <\text{s-expression}> | \text{true} | \text{false} \\
<\text{variable}> & ::= <\text{identifier}> \\
<\text{argument list}> & ::= <\text{argument}> | <\text{argument list}>; <\text{argument}> \\
<\text{argument}> & ::= <\text{form}> | [<\text{argument list}>] | <\text{constant}> | \text{undefined} \\
<\text{conditional list}> & ::= <\text{conditional}> | <\text{conditional list}>; <\text{conditional list}> \\
<\text{conditional}> & ::= <\text{form}> \rightarrow <\text{form}> \\
<\text{conditional expression}> & ::= [ <\text{conditional list}> ] \\
<\text{function}> & ::= \text{car} | \text{cdr} | \text{cons} | \text{atom} | \text{eq} | \text{null} | \text{eval} | \text{true} | \text{false} \\
& \quad | \{ \lambda (<\text{var list}) <\text{form}> \} \mid \text{def}[<\text{identifier}>] \mid <\text{identifier}> \\
<\text{s-expression}> & ::= <\text{atom}> | ( <\text{s-expression}>, <\text{s-expression}>) \\
<\text{atom}> & ::= <\text{identifier}> | \text{nil} \\
<\text{function definition}> & ::= \text{def}[<\text{identifier}>] = \{ \lambda (<\text{var list}) <\text{form}> \} \\
\end{align*}
\]

II. Rules of Inference

1. Rules for Eval

1.1 \text{eval}[ <\text{constant}> ] = <\text{constant}> \\
1.2 \text{eval}[ <\text{function}>[ <\text{argument list}>] ] = \\
\quad \text{ap}[ <\text{function}>; [\text{eval}[ <\text{argument list}>] ] ] \\
1.3 \text{eval}[ [ <\text{form}>_1 \rightarrow <\text{form}>_2 ] ] = \\
\quad \text{eval}[ \text{ap}[ \text{eval}[ <\text{form}>_1 ]; [ <\text{form}>_2 \text{ undefined } ]]] \\
\quad \text{eval}[ <\text{form}>_1 \rightarrow <\text{form}>_2; <\text{conditional list}> ] ] = \\
\quad \text{eval}[ \text{ap}[ \text{eval}[ <\text{form}>_1 ]; [ <\text{form}>_2; [ <\text{conditional list}> ] ] ]]
1.4 eval[ <argument list>; <argument>] =
    eval[ <argument list>]; eval[ <argument>]

2. Rules for Ap
   2.1 ap[ car; [(<s-expression>1,<s-expression>2)]] = <s-expression>1
   2.2 ap[ cdr; [(<s-expression>1,<s-expression>2)]] = <s-expression>2
   2.3 ap[ cons; [<s-expression>1; <s-expression>2]] =
           ( <s-expression>1 . <s-expression>2)
   2.4 ap[ null; [ <s-expression>]] = true if <s-expression> = nil
                else false †
   2.5 ap[ eq; [ <atom>1; <atom>2]] = true if <atom>1 =<atom>2
                else false †
   2.6 ap[ atom; [ <s-expression>]] = true if <s-expression> is
                <atom> else false †
   2.7 ap[ true; [ <form>1; <form>2]] = <form>1
   2.8 ap[ false; [ <form>1; <form>2]] = <form>2
   2.9 ap[ ff ; [ <constant>]] = eval[ def[ff] [<constant>]]

3. Any Instance of the Rule of \(\lambda\)-Application Applied to Phrases of
   the Lisp Syntax is a Rule of Inference
   \[
   3.0 \{\lambda(x) [ atom[x] \rightarrow x; \text{true} \rightarrow ff[\text{car}[x]]] \}[<constant>] =
   [ atom[<constant>] \rightarrow <constant>; \text{true} \rightarrow ff[\text{car}[<constant>]]]
   \]

4. Any Instance of a Function Definition is a Rule of Inference
   \[
   4.0 \text{def[ff]} = \{\lambda(x) [ atom[x] \rightarrow x; \text{true} \rightarrow ff[\text{car}[x]]]\}
   \]

† These rules are to be considered as schemata for the unbounded set
of instances of rules obtained by specifying the arguments of the
functions null, eq and atom in all possible ways and by computing
the results. These instances are preserved under \(T\) since \(T\) is 1-1
onto of the set Lisp data \(\cup\) \{true, false\} onto \(T[\text{Lisp data} \cup \{\text{true, false}\}].\)
III. A Trace for the evaluation of $ff([(a, b, c)]]$

begin eval [ $ff([(a, b, c)])$

1.2 ap[ $ff$ ; [eval[ ((a, b, c))]]

1.1 ap[ $ff$ ; [ ((a, b, c))]]

2.9 eval [ def$ff$ [(a, b, c)]]

4.0 eval [ {λ (x) [atom[x] → x ; true → ff[car[x]]]} [(a, b, c)]]

3.0 eval [ [atom[(a,b,c)]] → (a,b,c) ; true → ff[car[ ((a,b,c))]]]

1.3 eval [ ap [ eval[atom([(a, b, c))]]; [(a, b, c)]; [true→ff[car([(a, b, c))]]]]

1.2 eval [ ap [ ap[atom ; [eval[((a, b, c))]]]; [(a, b, c)]; [true→ff[car[((a, b, c))]]]]

1.1 eval [ ap[ap[atom ; [[((a, b, c))]]; [((a, b, c))]; [true→ff[car[((a, b, c))]]]]]

2.6 eval [ ap [false ; [(a, b, c) ; [true→ff[car[((a, b, c))]]]]]

2.8 eval [ [true → ff [ car [ ((a, b, c))]]]

1.3 eval [ ap [eval[true] ; [ff[car([(a, b, c))]]; undefined]]]

1.1 eval [ ap [ true ; [ ff[ car[((a, b, c))]]; undefined]]]

2.7 eval [ ff [ car [ ((a, b, c))]]

1.2 ap[ ff ; [ eval [ car [ ((a, b, c))]]]

1.2 ap[ ff ; [ ap [ car ; [ eval [ ((a, b, c))]]]]

1.1 ap[ ff ; [ ap [ car ; [ ((a, b, c))]]]

2.1 ap [ ff ; [ (a, b) ]]

2.9 eval [ def$ff$ [ (a, b) ]]

4.0 eval [ {λ (x) [atom[x] → x ; true → ff[car[x]]]} [(a, b)]

3.0 eval [ [atom[(a,b,c)]] → (a,b,c) ; true → ff[car[ (a,b,c)]]]

1.3 eval [ ap [ eval[atom[(a,b,c))]]; [(a, b) ; [true→ff[car[(a,b,c))]]]]

1.2 eval [ ap [ ap [ atom ; [eval[(a,b))]]; [(a, b)]; [true→ff[car[(a,b)]]]]]

1.1 eval [ ap [ ap [ atom ; [(a,b))]]; [(a, b); [true→ff[car[(a,b))]]]

2.6 eval [ ap [false ; [(a, b) ; [true→ff[car[(a,b))]]]]

2.8 eval [ [true → ff [ car [ (a,b) ]]]

1.3 eval [ ap [eval[true] ; [ff[car[(a,b))]]; undefined]]

1.1 eval [ ap [ true ; [ ff[car[(a,b))]]; undefined]]

2.7 eval [ ff [ car [ (a,b)]]]

1.2 ap [ ff ; [eval [ car [ (a, b)]]]]

1.2 ap [ ff ; [ap [ car ; [eval [ (a,b)]]]]]

1.1 ap [ ff ; [ap [ car ; [ (a, b)]]]]

2.1 ap [ ff ; [a ]]

2.9 eval [ def$ff$ [a ]]

4.0 eval [ {λ (x) [atom[x] → x ; true → ff[car[x]]]} [a ]]

3.0 eval [ [atom [ a] → a ; true → ff[car[a]]]]

1.3 eval [ ap [ eval [ atom [ a]] ; [ a ; [true → ff[car[a]]]]]]

1.2 eval [ ap [ ap [ atom ; [eval [ a]]]; [ a]; [true→ff[car[a]]]]]

1.1 eval [ ap [ ap [ atom ; [ a]]; [ a]; [true→ff[car[a]]]]]

2.6 eval [ ap [ true ; [ a ; [true→ff[car[a]]]]]]

2.7 eval [ a ]

1.1 a
The last line of this trace is, by definition, the value of the expression \texttt{eval [ ff [ ((a.b).c) ]]}. We now specify a mapping \(T\) on the Lisp Syntax I into a Formula Algol Syntax. \(T\) induces mappings on the rules of inference II and on the trace III. We note that \(T\), as defined below, satisfies the conditions required for \(T\) to be trace preserving, and in checking the translated trace we see that the trace is preserved. \(T\) is defined as follows:

\(T\) is the identity map except

1. On terminal characters of Lisp \(T\) is defined as
   
   \[
   \begin{align*}
   T \text{ car} & = \text{ leftoperand} \\
   T \text{ cdr} & = \text{ rightoperand} \\
   T \text{ cons} & = \text{ cons(BinaryFormula)} \\
   T \text{ atom} & = \text{ identifier} \\
   T \ [ & = ( \\
   T \ ] & = ) \\
   T \ ( & = [ \\
   T \ ) & = ]
   \end{align*}
   \]

2. On non-terminal characters of Lisp \(T\) is defined as
   
   \[
   \begin{align*}
   T \ <s\text{-expression}> & = <\text{formula}> \\
   T \ <\text{conditional list}> & = <\text{conditional part}> \\
   T \ <\text{argument}> & = <\text{parameter}> \\
   T \ <\text{argument list}> & = <\text{parameter list}>
   \end{align*}
   \]
3. On strings of Lisp, $T$ is defined as

\[
T (\text{<s-expression}>_1, \text{<s-expression}>_2) = [\text{<formula>}_1 - \text{<formula>}_2] \\
T \text{<form>}_1 - \text{<form>}_2 = \text{if } \text{<form>}_1 \text{then } \text{<form>}_2 \\
T [\text{<conditional list>}] = \text{<conditional part>} \\
T \text{<conditional>}; \text{<conditional list>} = \text{<conditional>\text{else} <conditional part>} \\
T \text{<argument list>}; \text{<argument>} = \text{<parameter list>}, \text{<parameter>} \\
T \text{ap} [\text{<function>}; [\text{<argument list>}] = \text{<function operator>} (\text{<parameter list>})
\]

Thus, having given $T$, we can give the translations of the syntax,

rules of inference and trace as follows:

I. T of the Syntax

\[
\text{<formula>} ::= [\text{<formula>} - \text{<formula>}] \mid \text{<atom>} \\
\text{<atom>} ::= \text{<identifier>} \mid \text{nil} \\
\text{<conditional>} ::= \text{if } \text{<form>} \text{then } \text{<form>} \\
\text{<conditional part>} ::= \text{<conditional>} \mid \text{<conditional> \text{else} <conditional part>} \\
\text{<conditional expression>} ::= \text{<conditional part>} \\
\text{<form>} ::= \text{<constant>} \mid \text{<variable>} \mid \text{<function operator>} (\text{<parameter list>}) \mid \text{<conditional expression>} \\
\text{<constant>} ::= \text{<formula>} \mid \text{true} \mid \text{false} \\
\text{<variable>} ::= \text{<identifier>} \\
\text{<parameter list>} ::= \text{<parameter>} \mid \text{<parameter list>}, \text{<parameter>} \\
\text{<parameter>} ::= \text{<form>} \mid (\text{<argument list>}) \mid \text{<constant>} \mid \text{undefined} \\
\text{<function operator>} ::= \text{leftoperand} \mid \text{rightoperand} \mid \text{cons(BinaryFormula)} \mid \text{identifier} \mid \text{eq} \mid \text{null} \mid \text{eval} \mid \text{true} \mid \text{false} \mid \text{<identifier>} \mid \text{def(<variable>)} \mid \text{eval (<form>)} \mid \{\lambda [\text{<var list>}] \text{<form>}\} \\
\text{<var list>} ::= \text{<variable>} \mid \text{<var list>}, \text{<variable>} \\
\text{<function definition>} ::= \text{def( <identifier> )} = \{\lambda [\text{<var list>}] \text{<form>}\}
\]
II. To the Rules of Inference

1. To the Rules for Eval

1.1 eval( <constant>) = <constant>

1.2 eval( <function operator> ( <parameter list>) ) = <function operator> ( eval( <parameter list>) )

1.3 eval( if <form>1 then <form>2 ) =
   eval( eval( <form>1)(( <form>2, undefined))

   eval( if <form>, then <form>2 else <conditional expression>) =
   eval( eval(<form>2)(( <form>2, <conditional expression>))

1.4 eval( < parameter list >, <parameter >) =
   eval( <parameter list >), eval( <parameter >)

2. To the Rules for Ap

2.1 leftoperand( [ <formula>1-<formula>2 ] ) = <formula>1

2.2 rightoperand([ <formula>1-<formula>2]) = <formula>2

2.3 cons(BinaryFormula)( <formula>1,<formula>2 ) =
   [ <formula>1 - <formula>2]

2.4 null ( <formula> ) = true if <formula> = nil else false

2.5 eq( <identifier>1, <identifier>2 ) = true if <identifier>1 =
   <identifier>2 else false

2.6 identifier ( <formula> ) = true if <formula> is <identifier>else false

2.7 true( <form>1, <form>2 ) = <form>1

2.8 false( <form>1, <form>2 ) = <form>2

2.9 ff( <constant> ) = eval( def(ff) ( <constant> ) )

3. To the Instances of λ-application

3.0 \{λ[x] if identifier(x) then x else if true then ff(leftoperand(x))\}
   ( <constant> ) = if identifier(<constant>) then <constant>
   else if true then ff(leftoperand(<constant>))

4. To the Instances of Function Definitions

4.0 def(ff) = \{λ[x] if identifier(x) then x else if true then ff(leftoperand(x))\}
III. T of the trace

begin eval ( ff ( [[a-b]-c] ))
1.2 ff ( eval ( [[a-b]-c] ))
1.1 ff ( [[a-b]-c] )
2.9 eval ( def(ff)( [[a-b]-c] ))
4.0 eval ( {λ [x] if identifier(x) then x else if true then ff(leftoperand(x)) } ( [a-b] ))
3.0 eval ( if identifier([[a-b]-c]) then [[a-b]-c] else if true then ff(leftoperand([[a-b]-c] ))
1.3 eval ( eval(identifier([[a-b]-c])) ( [a-b]-c ), if true then ff(leftoperand([[a-b]-c] )))
1.2 eval ( identifier( eval([[a-b]-c]) ) ( [a-b]-c ), if true then ff(leftoperand([[a-b]-c] )))
1.1 eval ( identifier([[a-b]-c]) ( [a-b]-c ), if true then ff(leftoperand([[a-b]-c] )))
2.6 eval ( false([[a-b]-c], if true then ff(leftoperand([[a-b]-c] )))
2.8 eval ( if true then ff(leftoperand([[a-b]-c] )))
1.3 eval ( eval(true) ( ff(leftoperand([[a-b]-c]]), undefined ))
1.1 eval ( true ( ff(leftoperand([[a-b]-c]]), undefined ))
2.7 eval ( ff(leftoperand([[a-b]-c]]))
1.2 ff ( eval ( leftoperand([[a-b]-c]]))
1.1 ff ( leftoperand ( [ a-b] )
2.1 ff ( [ a-b] )
2.9 eval ( def(ff)( [ a-b] )
4.0 eval ( {λ [x] if identifier(x) then x else if true then ff(leftoperand(x)) } ( [a-b] ))
3.0 eval ( if identifier([a-b]) then [a-b] else if true then ff(leftoperand([a-b] )))
1.3 eval ( eval(identifier([a-b])) ( [a-b], if true then ff(leftoperand([a-b] )))
1.2 eval ( identifier(eval([a-b])) ( [a-b], if true then ff(leftoperand([a-b] )))
1.1 eval ( identifier([a-b]) ( [a-b], if true then ff(leftoperand([a-b] )))
2.6 eval ( false([a-b], if true then ff(leftoperand([a-b] )))
2.8 eval ( if true then ff(leftoperand([a-b] )))
1.3 eval ( eval(true) ( ff(leftoperand([a-b])], undefined ))
1.1 eval ( true ( ff(leftoperand([a-b])], undefined ))
2.7 eval ( ff(leftoperand([a-b])]
1.2 ff ( eval ( leftoperand([a-b])]
1.2 ff ( leftoperand ( eval([a-b])]
1.1 ff ( leftoperand ( [a-b] )
2.1 ff(a)
2.9 eval ( def(ff)(a)
4.0 eval ( {λ [x] if identifier(x) then x else if true then ff(leftoperand(x))} (a)
3.0 eval ( if identifier(a) then a else if true then ff(leftoperand(a))

Discussion

We now propose to discuss why we think these two models are valid representations of Lisp evaluation and Formula Algol evaluation respectively. We do this by noting that the rules of inference in the first model have valid interpretations in Lisp and that the translation of the rules of inference from the Lisp model to the Formula Algol model yield rules of inference which are valid in a subset of Formula Algol.

Specifically, the rules of inference for eval are sequencing rules giving the Lisp order of evaluation of expressions in functional compositions, argument lists and conditional expressions. Rule 1.1 states that evaluation is the identity function on constants which is valid in Lisp and in Formula Algol. An interpretation of Rule 1.2 is that in function calls actual parameter lists are evaluated before calling a procedure. This is valid in Lisp and in Formula Algol provided all procedures used to model Lisp have formal parameters
called by value. [This is the case with the procedures used in the program in Chapter VI to represent Lisp in Formula Algol.]

Rule 1.3 asserts that conditional expressions are evaluated from left to right, which they are both in Lisp and in Formula Algol.

Rule 1.4 asserts that the evaluation of a list of actual parameters is the list of the evaluations of the actual parameters, and this is valid in Lisp and in Formula Algol (in Lisp actual parameters are called arguments).

The T translations of the set of Lisp identities specifying properties of primitive Lisp operators are all valid properties in Formula Algol. Assuming that the data structure definitions for formulae given in section 1 of Chapter IV define data structures of formulae in Formula Algol, where, for example, we assume that \([f-g]\) is an abbreviation for \(\text{cons}(\text{BinaryFormula})('-', f, g)\), the rule 2.1 is equivalent to \(\text{lefterand}(\text{cons}(\text{BinaryFormula})('-', f, g)) = f\) which is valid in the data definition facility since selectors are by definition the inverses of constructors. Rule 2.2 has a valid interpretation for the same reason. Rule 2.3 gives a definition of a notation. Rules 2.4, 2.5, and 2.6 have valid interpretations as Formula Algol predicates (directly definable in terms of predicates.
in the data definition facility). Specifically, eq( <identifier>\(_1\),<identifier>\(_2\))
is interpreted as the equality predicate, null(x) is interpreted as x='nil'
and identifier(x) is interpreted as x == identifier. Rules 2.7 and
2.8 are used as auxiliaries for expressing the order of evaluation
of conditionals, and help to specify that the evaluation of conditionals
is from left to right. Finally, rules 2.9, 3.0 and 4.0 express the nature
of a call of a function designator (by value) for ff (or for any function)
by specifying the substitution of actual for formal parameters in the
text which defines the body of the ff procedure, and by evaluating the
result. This, of course, has a valid interpretation in function
evaluation in Lisp and in procedure calls in Formula Algol with
parameters called by value.

Since (as we have shown) T preserves traces, any
expression e which is evaluated by giving its trace in the Lisp model
has an equivalent valid trace in the translation of this Lisp model
into the Formula Algol model. This shows that the Formula Algol
model accurately imitates the Lisp model in its evaluation of
expressions. In particular, since the result of evaluating a Lisp
expression is the value of the last line of its trace, and since the
last line of its trace maps under T into the last line of the translated
trace, we see that

\[
T[\text{LastLine}[\text{LispTrace}[\text{ff}(\text{a}.\text{b}.\text{c})]]] = \text{LastLine}[\text{FormulaAlgolTrace}[\text{Tff}(\text{T}(\text{a}.\text{b}.\text{c}))]]
\]

for the example above. But since \( T \) is invertible on data

\[
\text{LastLine}[\text{LispTrace}[\text{ff}(\text{a}.\text{b}.\text{c})]] = T^{-1}[\text{LastLine}[\text{FormulaAlgolTrace}[\text{Tff}([\text{a-b}]\text{-c})]]]
\]

and this is the basic equation (I) for the equivalence of evaluators \( M_1 \) and \( M_2 \). This equation, in fact, holds for the evaluation of all Lisp functions.

The technique we have presented in this chapter applies to the running Formula Algol program given in Chapter VI to show that it models the evaluation of Lisp functions faithfully, provided we accept that the Lisp and Formula Algol models accurately reflect the processes of evaluation in Lisp and Formula Algol respectively. The latter amounts to accepting that the rules of inference in the Lisp and Formula Algol models have valid interpretations in Lisp and Formula Algol respectively.
Chapter VI

Some Reduction Algorithms and Background Machines

The program presented in this chapter, written in Formula Algol [49], consists of a set of procedures which simulate both the data definition facility introduced in Chapter III, and several background machines. There are four stages of reduction among these background machines. First the data definition language defines data structures which are constructed from simulated plexes, (namely, blocks of k contiguous cells which may refer both to atomic data and to other blocks by a referencing scheme using simulated addresses associated with the blocks). Plexes are then simulated in Lisp. Lisp is simulated in Formula Algol with formula data structures, and Formula Algol is, of course, programmed in terms of the basic data structure of the G-21, a digital computer whose basic data structure is a linear array of 32-bit words. Hence, there is a four stage reduction pictured in figure 1 from defined data structures.

```
defined data structures
  ↓
plexes
  ↓
Lisp lists
  ↓
Formula Algol formulae
  ↓
G-21 words
```

figure 1
to the data structures of the actual computer. If we stop at a given stage of reduction R, chosen from the four stages of reduction above, and ignore the stages below, we see that we can define (by composition of reductions) a reduction from defined data structures to R. This shows that the machine at level R is a possible background machine for representing data structures defined with the data definition facility (although the representation may not be efficient). Hence, we have shown, for four examples, that the data structure definition facility is independent of background machine.

The Program

Some details of the simulation are as follows: Referring to the program at the end of the chapter, (1) Lines 18 through 25 define variables used in the program; (2) Lines 27 through 36 define patterns used as predicates over simulated descriptor formulae; (3) Lines 41 through 102 simulate the basic functions of Lisp in terms of Formula Algol where Lisp atoms are simulated by Formula Algol atoms, cons[x;y] is simulated by the formula x-y, and the special Lisp symbol nil is simulated by the atomic formula nil. The Lisp functions are seen to follow closely those found in Chapter IV, Section 2 discussing list processing.
(4) constructors  

Lines 170 to 232 give the functional which defines constructors from a simulated descriptor D and a possibly vacuous list of arguments L.

(5) predicates  

Predicates are given by the procedure Inst(Datum, D) which tests a Datum to see if it is a member of the data space described by the simulated descriptor D. Inst(Datum, D) is defined on lines 453 to 470 and uses the subroutine Pred found on lines 234 to 264.

(6) selection  

Selection is accomplished by the routine Select(x, y) found on lines 272 to 281, where x is a selection expression or is the name of a component to be selected, or is a path or path variable, and where y is a datum on which selection is to be performed.

(7) other procedures  

Other procedures simulate various forms of assignment (lines 314 to 346 and lines 376 to 387), administrate and convert between simulated values and locations (lines 283 to 312), file selector paths as associates of selector names in a table (lines 348 to 374), simulate the declaration of variables and allocation of space (lines 389 to 413), and simulate various special functions
of the data definition facility that calculate paths, determine types, and so forth (lines 415 to 451). The last several procedures print simulated data memory and simulated symbol tables.

(8) simulation of descriptors

Descriptor formulae are simulated by replacing operators within by arithmetic and Boolean operators of analogous precedence in Formula Algol formulae and by replacing their atoms with formula atoms. Computations on descriptors are then simulated by formula manipulation computations within Formula Algol. Patterns can be defined easily in Formula Algol to represent and to discriminate among classes of descriptors.

(9) interpretation

Formula Algol has the advantage that once a procedure is written this procedure immediately becomes an instruction in an interpreter, which interpreter is the eval routine. The call of a procedure may be represented as a formula and may be executed by applying eval to it. Thus formulae representing calls are an instruction format for an interpreter that executes the procedures previously defined. This principle is used in the construction of the simulation of the data definition facility.
BEGIN FORM DESCRIPTOR, TYPE, IDENTIFIER, ELNAME, NILL, DMEM, NVMEM, DECLARAND, INITIAL VALUE, INDEFINITE; SYMBOL DES, TYP, ELN, IDLIST; INTEGER DMAX;
FORM PROCEDURE TSMDBY(F, G); TSMDBY = TSMDBY(F, G);
FORM PROCEDURE SUCHTHAT(F, G); SUCHTHAT = SUCHTHAT(F, G);
FORM PROCEDURE DPR(X); DPR = DPR(X);
FORM PR, PT, NULLST, HEAD, TAIL, NAME, VALENCE, BONDLIST;
FORM COMPLEX, REALPART, IMAGPART; REAL R;

COMMENT; INITIALIZATIONS OF PATTERNS FOR DESCRIPTORS;
DES + [ANY-ANY, ANY-ANY, ANY-ANY, ANY-ANY, ANY-ANY, ANY-ANY];
DESCRIPTOR + OF (DES); TYP = [REAL, INTEGER, BOOLEAN, ATOM, OR, FORM, ANY];
TYPE + OF (TYP); IDLIST + [1]; IDENTIFIER + OF (IDLIST); ELN + [ATOM, DPR, (ANY)]; ELNAME + OF (ELN); DMAX + 1; DMEM + NILL; NVMEM + NILL; INITIAL VALUE + NILL.
BEGIN

COMMENTS: INNER BLOCK OF PROCEDURES;

BEGIN FORM X,Y,M,N,O,D;

FORM PROCEDURE CAR(F); VALUE F; FORM F;
  IF F :: X ANY - ANY THEN CAR-X ELSE
  BEGIN CAR-NIL; PRINT(.NO CAR); END;

FORM PROCEDURE CDR(F); VALUE F; FORM F;
  IF F :: ANY - XIANY THEN CDR-X ELSE
  BEGIN CDR-NIL; PRINT(.NO CDR); END;

FORM PROCEDURE CONS(F,G); VALUE F,G; FORM F,G;
  CONS :: F-G;

BOOLEAN PROCEDURE ATOMIC(F); VALUE F; FORM F;
  ATOMIC :: IF F :: ATOM THEN TRUE ELSE FALSE;

BOOLEAN PROCEDURE EQ(F,G); VALUE F,G; FORM F,G;
  EQ :: IF ATOMIC(F) :: ATOMIC(G) THEN
  BEGIN EQ :: IF F :: G THEN TRUE ELSE FALSE END ELSE
  PRINT(.EQ UNDEFINED);

BOOLEAN PROCEDURE NULL(F); VALUE F; FORM F;
  NULL :: IF F :: NIL THEN TRUE ELSE FALSE;

BOOLEAN PROCEDURE EQUAL(F,G); VALUE F,G; FORM F,G;
  EQUAL :: IF F :: G THEN TRUE ELSE FALSE;

INTEGER PROCEDURE LENGTH(L); VALUE L; FORM L;
  LENGTH :: IF NULL(L) THEN 0 ELSE 1 + LENGTH(CDR(L));

FORM PROCEDURE REVERSE(P); VALUE P; FORM P;
  BEGIN FORM R :: R-NIL;
  LOOP :: IF NULL(P) THEN GO TO EXIT ELSE
  R :: CAR(P)-R; P-CDR(P); GO TO LOOP;
  EXIT :: REVERSE-R;
  END;

FORM PROCEDURE APPEND(F,G); VALUE F,G; FORM F,G;
  APPEND :: REVERSE( CONS( G; REVERSE(F)));

BOOLEAN PROCEDURE MEMBER(F,G); VALUE F,G; FORM F,G;
  IF NULL(G) THEN MEMBER :: FALSE ELSE
  BEGIN IF EQUAL(F,CAR(G)) THEN MEMBER :: TRUE ELSE
  MEMBER :: MEMBER(F,CDR(G)); END;

FORM PROCEDURE SUBST(F,G,H); VALUE F,G,H; FORM F,G,H;
  IF ATOMIC(H) THEN BEGIN SUBST :: IF EQ(H,G) THEN F ELSE W END ELSE
  SUBST :: CONS(SUBST(F,G,CAR(H)),SUBST(F,G,CDR(H)));

FORM PROCEDURE ASSOC(X,A); VALUE X,A; FORM X,A;
  ASSOC :: IF EQUAL(CAR(CAR(A)),X) THEN CAR(A) ELSE ASSOC(X,CDR(A));

FORM PROCEDURE NTH(N,F); VALUE N,F; INTEGER N; FORM F;

END
REP \* TEMP;
END ELSE
PRINT(.NOREP);
END;

FORM PROCEDURE WITH(F,G); FORM F,G;
BEGIN FORM X,Y,Z,V,T,TEMP, MODIFIER;
TEMP \* F; MODIFIER \* G;
LOOP: IF MODIFIER \* Y(FORM \* ANY) \* NIL THEN
BEGIN
IF ZITEMP >> T(Y=ANY) THEN
WITH \* SUBS(.T)Z(X) ELSE WITH \* TEMP;
GO TO EXIT;
END;
IF MODIFIER \* X(YFORM \* ANY) \* VFORM THEN
BEGIN
IF ZITEMP >> T(Y=ANY) THEN TEMP += SUBS(.T)Z(X);
T \* .TJ MODIFIER \* V;
GO TO LOOP;
END;
EXIT;
END;

FORM PROCEDURE CONSTRUCT(D,L); VALUE D,L; FORM D,L;
BEGIN FORM X,Y,G,H,K,M;
IF D \* XIDESCRRIPTOR\*DESCRIPTOR THEN
BEGIN CONSTRUCT\*CONSTRUCT(X,L); GO TO EXIT10; END;
IF D \* ELNAME\*XIDESCRIPITOR THEN
BEGIN CONSTRUCT\*CONSTRUCT(X,L); GO TO EXIT10; END;
IF D \* DESCRIPTOR\*ANY THEN
BEGIN H\*NIL; K=D; M=L;
LOOP9: IF K \* XIDESCRRIPTOR\*Y|ANY THEN
BEGIN
G \* IF NULL(M) THEN CONSTRUCT(X,NILL) ELSE
CONSTRUCT(X,CONS(CAR(M),NILL));
H\*APPEND(H,G);K+X;IF \*NULL(M) THEN M\*EDR(M);
IF NULL(K) THEN GO TO NEXT1 ELSE GO TO LOOP9;
NEXT1: CONSTRUCT\*DMAX; DMAX\*DMAX+1;
DMEM\*APPEND(DMEM,H); GO TO EXIT10;
END;
END9;
IF D \* ATOM \* XITYPE THEN
BEGIN CONSTRUCT\*CONSTRUCT(X,L); GO TO EXIT10; END;
IF D \* EQ.(X|ANY) THEN
BEGIN
IF \*NULL(L) THEN X\*CASK(L);
M\*X \* REAL THEN REAL! ELSE
IF X \* INTEGER THEN INTEGER ELSE
IF X \* BOOLEAN THEN BOOLEAN ELSE
IF X \* ATOM THEN ATOM ELSE
IF X \* FORM THEN FORM! ELSE ANY;
H \* CONS(M,CONS(X,NILL));
CONSTRUCT \* DMAX; DMAX\*DMAX+1;
DMEM\*APPEND(DMEM,H); GO TO EXIT10;
END;
IF D == DPR.(X:FORM) THEN
    BEGIN H- EVAL X; CONSTRUCT-CONSTRUCT(H,L); GO TO EXIT10; END
IF D == IDENTIFIER THEN
    BEGIN CONSTRUCT-CONSTRUCT(CAR(CDR(assoc(D,NVEM))),L);
        GO TO EXIT10; END
IF D == SUCHTHAT.(X:FORM, ANY) THEN
    BEGIN CONSTRUCT-CONSTRUCT(X,L); GO TO EXIT10; END
IF D == TSP:DRY.(X:FORM, Y:FORM) THEN
    BEGIN CONSTRUCT-APPLY(Y, CONSTRUCT(X,L)-NILL));
        GO TO EXIT10; END
IF D == WITH.(ANY, ANY) THEN
    BEGIN CONSTRUCT-CONSTRUCT(EVAL D, L); GO TO EXIT10; END
IF D == REP.(X:ANY, Y:ANY) THEN
    IF X == INDEFINITE THEN
        BEGIN CONSTRUCT-CONSTRUCT(REP.(LENGTH(L), Y), L);
            GO TO EXIT10; END
    BEGIN CONSTRUCT-CONSTRUCT(EVAL D, L); GO TO EXIT10; END
IF D == TAG.(X:ANY) THEN
    BEGIN CONSTRUCT-CONSTRUCT(EVAL D, L); GO TO EXIT10; END
IF D == NIL THEN
    BEGIN H- CONST(ANY, CONST(NULL, NULL));
        CONSTRUCT-DMAX; DMAX+1; DMEM-APPEND(DMEMH);
        GO TO EXIT10;
    END
    IF D == X:TYPE THEN
        BEGIN M- IF NULL(L) THEN: NULL ELSE CAR(L);
            H- CONST(CONS(M, NILL)); CONSTRUCT-DMAX;
            DMAX-DMAX+1; DMEM-APPEND(DMEMH); GO TO EXIT10;
        END
EXIT10:
END;

BOOLEAN PROCEDURE PRED(D, DATUM, N); VALUE D, DATUM, N;
FORM D, DATUM; INTEGER N;
BEGIN FORM X, Y, G; INTEGER I; SYMBOL T;
IF D == IDENTIFIER THEN
    BEGIN PRED- PRED(CAR(CDR(assoc(D,NVEM))), DATUM, N);
        GO TO EXIT11; END
IF D == X:DESCRIPTOR-Y:DESCRIPTOR THEN
    BEGIN PRED-PRED(X, DATUM, N)-PRED(Y, DATUM, N); GO TO EXIT11; END
IF D == ELNAME+X:DESCRIPTOR THEN
    BEGIN PRED-PRED(X, DATUM, N); GO TO EXIT11; END
IF D == X:DESCRIPTOR=NILL THEN
    BEGIN G=NTH(EVAL DATUM, DMEM); G- NTH(N, G);
        PRED- PRED(X, G, 1); GO TO EXIT11; END
IF D == X:DESCRIPTOR-Y:DESCRIPTOR THEN
    BEGIN G- NTH(EVAL DATUM, DMEM);
        G- NTH(N, G); PRED- PRED(X, G, 1)-PRED(Y, DATUM, N+1);
        GO TO EXIT11; END
IF D == IDENTIFIER-X:TYPE THEN
    BEGIN PRED-PRED(X, DATUM, N); GO TO EXIT11; END
IF D == EQ.(X:ANY) THEN
    BEGIN G- NTH(EVAL DATUM, DMEM); T- [CAR(G)];
        IF AMONG(T, TYP) THEN PRED- CAR(CDR(G))==X ELSE
            PRED- FALSE; GO TO EXIT11; END
IF D == DPR. (X:FORM) THEN
BEGIN G = EVAL X; PRED + RED (G, DATUM); GO TO EXIT11; END;
IF D == TYPE THEN
BEGIN G = NTH(EVAL DATUM, DMEM); T = (CAR(G));
IF AMONG(T, TYP) THEN PRED + CAR(CDR(G)) = D ELSE
PRED + FALSE; GO TO; EXIT11; END;
EXIT11; 
END;

FORM: PROCEDURE CONTENTS(L); VALUE: L; FORM: L;
BEGIN FORM X, Y, J; INTEGER I, IJ;
X = CAR(L); Y = CAR(CDR(L)); I = EVAL X; J = EVAL Y; Z = O;
CONTENTS = IF J = O THEN L ELSE CONS(NTH(J, NTH(I, DMEM)), CONS(Z, NIL));
END;

FORM: PROCEDURE SELECT(X, Y); VALUE: X, Y; FORM: X, Y;
BEGIN FORM T;
IF X == IDENTIFIER THEN X = CAR(CDR(assoc(X, NVMEM)));;
IF Y == IDENTIFIER THEN T = CONTENTS(CAR(CDR(assoc(Y, NVMEM)))) ELSE TRY;
IF NULL(X) THEN SELECT + T ELSE:
BEGIN
IF NULL(CDR(X)) THEN SELECT + CONS(CAR(T), CONS(CAR(X), NIL)) ELSE
SELECT + SELECT(CDR(X), CONTENTS(CONS(CAR(T), CONS(CAR(X), NIL))));
END;
END;

FORM: PROCEDURE DATERTERM(V); FORM: V;
BEGIN FORM X;
X = CAR(CONTENTS(V));
X = NTH(EVAL X, DMEM);
DATERTERM = CAR(CDR(X));
END;

FORM: PROCEDURE LEFTHANDVALUE(E); VALUE: E; FORM: E;
BEGIN FORM X, Y;
IF E == IDENTIFIER THEN
BEGIN LEFTHANDVALUE = CAR(CDR(assoc(E, NVMEM))); GO TO EXIT14; END;
IF E == SELECT.(ANY, ANY) THEN
BEGIN LEFTHANDVALUE = EVAL E; GO TO EXIT14; END;
PRINT (NOLHV);
EXIT14; 
END;

FORM: PROCEDURE RIGHHANDVALUE(E); FORM: E;
BEGIN FORM X;
IF E == CONSTRUCT.(ANY, ANY) THEN RIGHHANDVALUE = EVAL E ELSE
RIGHHANDVALUE = CAR(CONTENTS(LEFTHANDVALUE(E)));
END;

INTEGER: PROCEDURE POSITION(X, Y); VALUE: X, Y;
BEGIN FORM P; INTEGER I; P = Y; I = 1;
LOOP10: IF NULL(P) THEN BEGIN POSITION + 0; GO TO EXIT15; END;
IF EQUAL(X, CAR(CAR(P))) THEN
BEGIN POSITION = I; GO TO EXIT15; END;
I = I + 1; P = CDR(P); GO TO LOOP10;
EXIT15; 

PROCEDURE NVASSIGN(I, V) VALUE I, V; FORM I, V;
BEGIN INTEGER K; K = POSITION(I, NVMEM);
NVMEM = IF K=0 THEN CONS(CONS(I, CONS(V, NIL)), NVMEM) ELSE
ALTRK(NVMEM, CONS(I, CONS(V, NIL)));
IF NOTAMONG(I, IDLIST) THEN INSERT (I) AFTER LAST OF IDLIST;
END;

PROCEDURE DASSIGN(L, V) VALUE L, V; FORM L, V;
BEGIN FORM F; INTEGER I, J;
F = CAR(L); I = EVAL F; F = CAR(CDR(L)); J = EVAL F;
IF J = 0 THEN DMEM = ALTR(I, DMEM, CONS(V, NIL)) ELSE
DMEM = ALTR(I, DMEM, ALTR(J, DMEM, V));
END;

PROCEDURE NORMASGN(E1, E2) VALUE E1, E2; FORM E1, E2;
BEGIN FORM X, Y, Z, W; Z = 0;
IF E1 = IDENTIFIER THEN
NVASSIGN(E1, CONS(RIGHTHANDVALUE(E2), CONS(Z, NIL)));
ELSE
BEGIN
IF E2 = SELECT.(ANY, ANY) THEN
BEGIN W = EVAL E2; DASSIGN(W, RIGHTHANDVALUE(E2)) END;
END;
END;

PROCEDURE LOCASGN(E1, E2) VALUE E1, E2; FORM E1, E2;
NVASSIGN(E1, LEFTHANDVALUE(E2));

PROCEDURE OVRASGN(E1, E2) VALUE E1, E2; FORM E1, E2;
BEGIN FORM F; INTEGER I, J;
F = RIGHTHANDVALUE(E1); I = EVAL F; F = RIGHTHANDVALUE(E2); J = EVAL F;
DMEM = ALTR(I, DMEM, COPI(NTH(J, DMEM)));
END;

PROCEDURE FILESELECTORS(D, P; N) VALUE D, P, N;
FORM D, P; INTEGER N;
BEGIN FORM X, Y;
IF D = SUCHTHAT(X; FORM, ANY) THEN
BEGIN FILESELECTORS(X, P; N) GO TO EXIT12; END;
IF D = TSFMDBY.(X; FORM, ANY) THEN
BEGIN FILESELECTORS(X, P; N) GO TO EXIT12; END;
IF D = WITH.(ANY; ANY) THEN
BEGIN FILESELECTORS(EVAL(D, P; N)) GO TO EXIT12; END;
IF D = TAG.(ANY) THEN
BEGIN FILESELECTORS(EVAL(D, P; N)) GO TO EXIT12; END;
IF D = REP.(ANY, Y; ANY) THEN
BEGIN FILESELECTORS(Y, P; N) GO TO EXIT12; END;
IF D = X; DESCRIBER; Y; DESCRIBER THEN
BEGIN FILESELECTORS(X, P; N) FILESELECTORS(Y, P; N) GO TO EXIT12; END;
IF D = ELNAME; X; DESCRIBER THEN
BEGIN FILESELECTORS(X, P; N) GO TO EXIT12; END;
IF D = X; DESCRIBER; Y; DESCRIBER THEN

BEGIN FILESELECTORS(X,N-P,1); FILESELECTORS(Y,P,N+1);
GO TO EXIT12; END;
IF D = XIDESCRIPTOR-NILL THEN
BEGIN FILESELECTORS(X,N-P,1); GO TO EXIT12; END;
IF D = XIDATOM - ANY THEN
BEGIN NVASSIGN(X,REVERSE(P)); GO TO EXIT12; END;
EXIT12;
END;

PROCEDURE IMPLICITASSIGN(D); VALUE D; FORM D;
BEGIN FORM X,Y;
IF D = XIDESCRIPTOR-YIDESCRIPTOR THEN
BEGIN IMPLICITASSIGN(Y); IMPLICITASSIGN(Y); GO TO EXIT13; END;
IF D = XIELNAME + YIDESCRIPTOR THEN
BEGIN NVASSIGN(X,Y); GO TO EXIT13; END;
EXIT13;
END;

PROCEDURE DESCRASSIGN(E1,E2); FORM E1,E2;
BEGIN FILESELECTORS(E2,NILL,1);
NVASSIGN(E1,E2); IMPLICITASSIGN(E2); GO;
END;

PROCEDURE DECLARE(D,L); VALUE D,L; FORM D,L;
BEGIN FORM X,Y; Z=0;
LOOP11 IF NULL(L) THEN GO TO EXIT17 ELSE
BEGIN
IF D = XIDESCRIPTOR - DESCRIPTOR THEN
BEGIN DECLARE(X,L); GO TO EXIT16; END;
IF D = ENAME + XIDESCRIPTOR THEN
BEGIN DECLARE(X,L); GO TO EXIT16; END;
IF D = DPR.(X1ANY) THEN
BEGIN DECLARE(EVAL X,L); GO TO EXIT16; END;
IF D = IDENTIFIER THEN
BEGIN DECLARE(CAR(COR(ASSOC(D,NVMEM))),L)
GO TO EXIT16; END;
IF NULL(INITIALVALUE) THEN
BEGIN G-CONST(CONSTRUCT(D,NILL),CONSTZ,NILL));
NVASSIGN(CAR(L),GO; GO TO EXIT16; END;
ELSE BEGIN
IF INITIALVALUE == DECLARAND THEN
BEGIN NVASSIGN(CAR(L),CAR(L)); GO TO EXIT16; END;
ELSE BEGIN NVASSIGN(CAR(L),INITIALVALUE)); GO TO EXIT16; END;
END;
END;
EXIT16;
L = CDR(L); GO TO LOOP11;
EXIT17;
END;

INTEGER PROCEDURE TIPE1(DA,D,N); VALUE DA,D,N;
FORM DA,D; INTEGER N;
BEGIN FORM X,Y;
IF D = XIDESCRIPTOR-YIDESCRIPTOR THEN
BEGIN IF PRED(X,DA,1) THEN TIPE1-N ELSE
TIPE1 = TIPE1(DA,Y,1); END;
IF D = XIDESCRIPTOR THEN
BEGIN IF PRED(X,DA,1) THEN TIPE1=N ELSE
  TIPE1=0 END;
END;

INTEGER PROCEDURE TIPE(DATUM,D) VALUE DATUM,D FORM DATUM,D;
TIPE = TIPE1(DATUM,D,1));

FORM PROCEDURE PATH1(D1,D2,R4) VALUE D1,D2,P,N);
FORM D1,D2,P; INTEGER N;
BEGIN FORM X,Y,G
  IF D2 = D1 THEN BEGIN PATH1=REVERSE(P);GO TO EXIT1; END;
  IF D1 = XIDESCRPTOR* YIDESCRPTOR THEN
    BEGIN IF X >> D2 THEN PATH1=PATH1(X&D2,Y,P,N) ELSE
      PATH1=PATH1(Y,D2,P,V) GO TO EXIT1; END;
  IF D1 = XIDESCRPTOR* YIDESCRPTOR THEN
    BEGIN PATH1=PATH1(X,D2,R4,N) GO TO EXIT1; END;
  IF D1 = XIDESCRPTOR THEN
    BEGIN PATH1=PATH1(X,D2,R4,N) GO TO EXIT1; END;
  BEGIN G = N;
    IF X >> D2 THEN PATH1=PATH1(X,D2,CONS(G,P),1)
      ELSE PATH1=PATH1(Y,D2,P,N+1);
    GO TO EXIT1;
END;
IF D1 = IDENTIFIER THEN X:ANY THEN
  BEGIN PATH1=PATH1(X,D2,N); GO TO EXIT1; END;
IF D1 = NULL THEN PRINT(NOPATH); PATH1=NILL
EXIT1; 
END;

FORM PROCEDURE PATH(D1,D2) VALUE D1,D2 FORM D1,D2;
PATH = PATH1(D1,D2,NILL,1));

BOOLEAN PROCEDURE INST(DATUM,D) VALUE DATUM,D FORM DATUM,D;
BEGIN BEGIN FORM X,Y,B
  IF D = SUCHTHAT(X FORM,Y FORM) THEN
    BEGIN B=APPLY(Y,DATUM-NILL); INST=INST(DATUM,X) EVAL B END;
  IF D = TSFMDRY(X FORM,ANY) THEN
    INST = INST(DATUM,X);
  IF D = WITH,(ANY,ANY) THEN
    INST = INST(DATUM,EVAL C);
  IF D = TAG,(ANY) THEN
    INST = INST(DATUM,EVAL C);
  IF D = REP,(IANY,YIANY) THEN
    INST = IF X = INDEFINITE THEN INST(DATUM,REP*(LENGTH(DATUM),Y))
      ELSE INST(DATUM,EVAL D);
  IF DATUM = IDENTIFIER THEN
    BEGIN DATUM=CAR(CDR(ASSOC(DATUM,NVMEM)));)
    DATUM = CAR(VALUES(DATUM)); END;
  INST = PREX(DATUM,1);
END;

PROCEDURE PDMEM;
BEGIN INTEGER N; FORM F; SYMBOL S,T;
N=1; F=DMEM;
LOOP; IF NULL(F) THEN GO TO EXIT ELSE
  T = L(CAR(F)); S = [N,T]; PRINT(S)
END;
F = CDR(F); N = N + 1; GO TO LOOP;
EXIT;
END;

PROCEDURE PRNVMEM;
BEGIN FORM F; SYMBOL T; F = VVMEM;
LOOP: IF NULL(F) THEN GO TO EXIT ELSE
T = L(CAR(F)); PRINT(T); F = CDR(F);
GO TO LOOP;
EXIT;
END;

BEGIN FORM V,W,Z;
   READ 55212 1
   PRINT(111); R+;
   DESCRASGN(COMPLEX, (REALPART-REAL) * (IMAGPART-REAL); NILL));
   GC;
   DECLARE(COMPLEX:CONS(Z,NILL));
   NORMASGN(Z,CONSTRUCT,(COMPLEX, (3.5-(4.2-NILL)) ));
   V = SELECT(REALPART,Z);
   PRINT(7777); PRINT(DATATERM(V));
   IF INST(Z,COMPLEX) THEN PRINT(1234) ELSE PRINT(4321);
   PRINT(8888); PRNVMEM; GC;
   PRINT(9999); PRDVMEM;
   REMO 1
END; END; END;
BEGIN EXECUTION 22:01:09; 04379 AVAILABLE CELLS:

1111
7777
(-35000000000+01)
1234
8888
IZ, [6, 0])
[COMPLEX, (REALPART - REAL) * (IMAGPART - REAL) + NILL]
[IMAGPART, [2]]
[REALPART, [1]]
9999
[1, (REAL, NILL)]
[2, (REAL, NILL)]
[3, [1, 2]]
[4, (REAL, (.35000000000+01))]
[5, (REAL, (.42000000000+01))]
[6, [4, 5]]

TIME USED: 00:02:15 PAGES: 10 22374 22:01:29 0 9 0 2 0 27 0 25 0
FORM PROCEDURE SEQUENCE(N,T); VALUE N,T; FORM N,T;
BEGIN INTEGER I,J; FORM TEMP; J=EVAL N; TEMP=NULL;
FOR I = 1 STEP 1 UNTIL J DO TEMP = T*TEMP;
SEQUENCE = TEMP;
END;

FORM PROCEDURE PAIR(T); FORM T;
BEGIN
REOR 37341 00037347
PAIR = SUBS(.P,T)PR(T); END;

FORM PROCEDURE LIST(T); VALUE T; FORM T;
LIST = NULLLIST*EQ.(NILL) + DPR.(PAIR.(T)) +
(HEAD-T) + (TAIL-DPR.(LIST(T)))*NILL);

FORM PROCEDURE ATOMM(N); VALUE N; FORM N;
ATOMM = (NAME-ATOM)*((VALENCE-EG.(N)) +
(BONDLIST-DPR.(SEQUENCE.(N,ANY))*NILL));

PR+((HEAD+PT)+T)(TAIL-DPR.(LIST.(PT)))*NILL);;

22027
V+4;

W+ CONSTRUCT( ATOMM(V), NILL));
PRINT(W,1111); GC;
W+CONSTRUCT(SEQUENCE(V,REAL),(2-(3-(4-(5-NILL)))));)
PRINT(W,2222); GC;
W+ CONSTRUCT(LIST(REAL),NILL);
PRINT(W,3333); GC;
W+CONSTRUCT(DPR.(SEQUENCE.(V,REAL)),NILL);
PRINT(W,4444); GC;
W+CONSTRUCT(PAIR(REAL), (3-NILL));
PRINT(W,5555); GC; PROMEM;
REMO 1
END;END;END;
BEGIN EXECUTION 23:55:48: 04407 AVAILABLE CELLS

1111
13
2222
14
3333
19
4444
22
5555

[1, [ATOM, NILL]]
[2, [INTE, 4]]
[3, [ANY, NILL]]
[4, [ANY, NILL]]
[5, [ANY, NILL]]
[6, [ANY, NILL]]
[7, [3, 4, 5, 6]]
[8, [1, 2, 7]]
[9, [REAL, 2]]
[10, [RBAL, 3]]
[11, [RBAL, 4]]
[12, [RBAL, 5]]
[13, [9, 10, 11, 12]]
[14, [ATOM, NILL]]
[15, [RBAL, NILL]]
[16, [RBAL, NILL]]
[17, [RBAL, NILL]]
[18, [RBAL, NILL]]
[19, [15, 16, 17, 18]]
[20, [RBAL, 3]]
[21, [ATOM, NILL]]
[22, [23, 21]]

TIME USED: 00:02:52 PAGES: 10 22305 23:56:21 34 0 0 0 6 0 0 0 44 0
Chapter VII

Future Directions and Conclusions

1. Combining the Data Definition Facility with a Syntax Definition Facility

Once a data definition facility has been added to a language, it follows that its syntax should be kept variable or open-ended, for once operations and data structures are defined well adapted to a task it should be possible to define notation evoking these operations that is well adapted also. This is particularly so when we are dealing with objects for which "a commonly used notation exists, in which case we should be able to use such a notation in our programming languages with the same meaning [61]''. Accepted notations for dealing with mathematical objects provide many illustrations. For example, arithmetic infix notation is a commonly used notation for objects such as matrices, complex variables, integers and real numbers, chosen from fields or rings. In a programming language dealing simultaneously with several kinds of these objects, it should be possible to share a common syntax for arithmetic infix notation. A recent proposal of Galler and Perlis [24] provides a compact method for doing this.

The data definition facility supplies a fixed notation for predicates, selectors, constructors and the introduction of new
operations (procedure notation). This standard notation is fixed at the time the system is defined, it must be used for all applications, and it cannot be varied. This soon leads to situations with which we are not willing to live. We then feel compelled to introduce more concise notations for purposes of clarity and economy of expression. In several places in this dissertation we have surreptitiously defined and used abbreviated notations for various constructions, although a syntax definition facility permitting us to do so was never specified formally. For example, \( a + b \) was used as a contraction for \( \text{cons} \left( \text{BinaryFormula} \left( '+', a, b \right) \right) \), \( a \uparrow b \) was used as a contraction for \( \text{cons} \left( \text{ParallelElement} \left( a, b \right) \right) \) and \( (a, b) \) was used as a contraction for \( \text{cons} \left( \text{Pair} \left( a, \text{cons} \left( \text{Pair} \left( b, \text{cons} ( \text{Nullist} ) \right) \right) \right) \right) \). A reward for indulging in this lack of rigor is that we are in a position to make comparisons and to expose issues. The reader is invited to observe for himself the magnitude of the degradation in clarity and conciseness that results from interchanging these contractions with their equivalents (see for example the presentations on pp. 125-127). After observing the difference he will probably agree that adding a syntax definition facility to a language to which we have added a data definition facility constitutes not merely a minor aesthetic improvement, but rather a compelling necessity. In short, having variable data structures implies having variable syntax.
Fortunately, several methods for providing languages with variable syntax are well known, and while it is beyond the scope of this thesis and would weaken it to specify a particular technique for coupling a syntax definition facility with the data definition facility, no extraordinary stretch of the imagination is required to see how this could be done. One technique for providing variable syntax, for example, is demonstrated in the Compiler Compiler of Brooker and Morris [11, 53] which has been shown to be an adequate solution on the basis of experience gained with several operating systems. Brooker reported in [10], for example, that a Compiler Compiler version of Atlas Autocode (essentially a dialect of Algol) ran only two times slower than a hand coded version. Brooker and Morris represent source language syntax in a notation which is similar in intent to Backus Normal Form [3] but which contains additional punctuation specifying whether syntactic constructions are to be regarded as repeating indefinitely or optionally present. A top down, slow back analyzer, which operates interpretively from a coded form of the syntax notation, parses source language into what Brooker and Morris call "analysis records" which are phrase structures of the input sentences with respect to the grammar of the source language. Analysis records are then reduced to machine
code by hierarchical calls on a set of generators. The syntax may be extended by writing new "formats" for input sentences in the syntax notation. Another system described by Ingerman [28] uses a bottom up analyzer to achieve open ended syntax. A third method of considerable efficiency, permitting the syntax analyzer to be compiled for the initial version of the source language, and permitting syntax extension by differential compilation of the recognizer may be possible and is under investigation by Earley [16]. Finally, as we have already noted, Galler and Perlis [24] have given a compact technique for making extendable the class of operators, operands and operator infix expressions in Algol and have treated with particular effectiveness the problem of how to share the syntax of arithmetic expressions among several different data types. Their technique for providing the semantics of newly defined operations is to use macro expansions.

As is customary in other macro systems, the processing of definitions proposed here follows a simple pattern: Take a language x, extend its syntax to a language x' to include that syntax necessary for phrasing and using definitions, and then reduce a text in the extended syntax to an "equivalent" one in language x. [24 , p. 204].

We should remark that there is more to the Galler and Perlis approach than the introduction of a variable syntax for expressions.
In fact, it constitutes an alternative approach to part of the work of this thesis. Using arrays to represent composite data structures, it allows one to define operations on new composite data types by giving an operator syntax and specifying how this syntax reduces to an equivalent syntax composed from the currently defined syntax of the parts of the composite data. Thus it demonstrates that using arrays and syntax expansion in this fashion provides a synthetic tool for introducing new data types. The use of this synthetic tool for defining new data types is explored in [24] by exhibiting four application areas (matrices, lists, files and complex variables) which are in common with a subset of those given in this dissertation. The exercise of comparing the examples in these application areas in these two approaches in order to determine if a common principle is at work leads to the following observations.

All definition facilities in programming share an important feature: the definition of an object need be given only once prior to the scope of its application in which it may be invoked arbitrarily many times, and the construction used to specify the definition need be given only once permitting the use of a concise equivalent in place of repeated use of the defining construction. This helps produce
clarity of structure and can produce important savings both in
the reading and in the writing of programs. A facility for defining
procedures (or, equivalently, routines or functions) is acknowledged
to be basic in programming because it permits the rule for a
computation to be given only once in order to apply it an arbitrary
number of times, making it unnecessary to repeat the construction
for the rule in each application. The data definition facility
presented in this dissertation allows the definition of data structures
to be expressed once prior to writing processes over the defined data
structures, instead of programming these processes indirectly at all
times in terms of a fixed set of initial data primitives. A syntax
definition facility permits burdensome constructions to be replaced
by concise equivalents defined once prior to all applications. We
see as a main contribution of the Galler and Perlis paper [24] that it
exposes the point of view that once definition facilities for data and
syntax are given together, then, in defining operations on composite
data structures, we can often make good use of the inherent syntax of
the parts. For example, in their paper [24, p. 215] complex
multiplication is defined as follows:

```plaintext
complex means array [1:2];

. . .
complex a × complex b := complex ' complex
```
Here we see that the syntax of real numbers is used in giving the expressions for the real and imaginary parts of the product. The same principle is used implicitly in places in this dissertation. For example, in giving the definitions of the complex product of \( u \) and \( v \) (pages 19 and 128) the Algol infix syntax of real numbers is used on the real and imaginary parts of \( u \) and \( v \) to give the expressions for the real and imaginary parts of the product. By contrast, the definition of matrix multiplication (p.135) uses functional notation (i.e. procedure notation) to specify the expressions for the elements of the product matrix, and this though somewhat more general is less natural. To summarize, even though functional notation is always applicable, using a defined or a natural notation of the parts of a composite object often improves the clarity of a definition of an operation on it.

Open - Ended Systems

An important consequence of introducing syntax and data definition facilities is that we are lead naturally to the design of open-ended systems. If we can produce an open-ended system in which it is as cheap to define new notations and new data spaces as it is to define procedures in present day systems, then we have changed the economics of programming in a significant way because
it then becomes cheap (both in complexity of design and in composition time) for a programmer to introduce special purpose notations and data spaces. Perlis has defined the economics of the situation in his Turing Lecture [48] as follows:

We know that we design a language to simplify the expression of an unbounded number of algorithms created by an important class of problems. The design should be performed only when the algorithms for this class impose, or are likely to impose, after some cultivation, considerable traffic on computers as well as considerable composition time by programmers using existing languages. The language, then, must reduce the cost of a set of transactions to pay its cost of design, maintenance, and improvement.

It follows from this that if we reduce significantly the costs of the "design, maintenance and improvement" of languages the amount of use needed to justify the creation of a language is reduced correspondingly. In particular, reducing significantly the composition time and programming skill required to levels that a non-professional user can afford and command, will mean that the creation of personalized languages and libraries of notations becomes feasible. A notation library may, for example, contain a package for formula manipulation, a package for list processing, a package for matrix algebra, a package for complex arithmetic, and so on. For a series of applications a user could specify which of the several notational
packages he desires to include in a language and he could define extensions if necessary. In doing so he would have assembled a language suited to his needs and he would not have paid time and space for generality he didn't need and couldn't use.

For users characterized by two properties: (1) they have esoteric computing requirements for which no currently available language is satisfactory, and (2) they don't have the resources to implement an appropriate language, the economic change offered by open-ended systems should represent a welcome advance in programming technique. A main contribution of this thesis is, I believe, that it gives one way to supply one of the missing pieces needed to build versatile and effective open-ended systems; systems not only versatile with regard to the syntax they can accept but with regard to the range of different data spaces and operations that such syntax can describe and evoke with ease.

Conclusions

This dissertation has presented a notation for data structures intended to summarize and to integrate known structures, to have versatile descriptive power, and to lead naturally to the description
of new structures. By embedding this notation in a programming language both a new synthetic tool and a new programming technique have been offered, and the use of this synthetic tool and this programming technique has been explored in a number of application areas. It is hoped that the examples of the uses of the data definition facility have shown several things, among which are: that a value of defining data structures explicitly and formally is that it leads to conciseness and clarity of structure in the definition of associated processes, that it permits the description of data and processes at the full level of sophistication and complexity demanded by existing languages of practical utility (here we have in mind Formula Algol and Lisp), and that it provides a method for improving significantly the versatility and generality of present day programming languages with regard to the data spaces with which they can deal gracefully. We also see as one strength of the data definition facility that it focuses on a missing piece necessary for the construction of open-ended systems. Finally, the formal study of the equivalence of evaluators and their data structures, in Chapter V, was intended as a hopeful sign that giving a data structure notation of broad descriptive power provides a helpful precedent for the formalization of such notions as the behavioral equivalence of different data representations.
There are many important problems which we confess are beyond the scope of this thesis. Among these are problems of efficiency and optimization among equivalent data structures and problems of representation theory such as stating constraints associated with programming tasks and then finding or generating representations that satisfy them.

This thesis has presented only one approach to the study of data structures, which has been to provide static data descriptions and to associate with these static descriptions three classes of operators: predicates, selectors and constructors. The set of processes over the described structures is then conceived to be a subset of the set we get by taking the closure of the set of predicates, selectors and constructors under a given class of composition rules for writing programs. The higher order processes we need are then specified by writing procedures. In the case that a syntax definition facility is also present we may introduce appropriate notations whose semantics may be given in terms of these defined processes.

Another view of data description holds that in addition to or instead of giving static data descriptions we should specify for each data structure introduced a set of behaviors permissible. For example,
in defining a stack we specify that elements may be added, deleted and accessed only at one place (called its top) and nowhere else. Thus, given that we can represent stacks using, for example, vectors or lists as the underlying static structure, what defines a stack is its accessing law and its growth and decay law, not the static structure used to model it. Another example is the distinction between an integer and a date. Dates (such as 1876 and 1932) and integers may be represented by the same static structure, but they are distinguished by their combining and comparison laws (for example, dates may be compared but not multiplied whereas integers may be both compared and multiplied). Thus, this view of data structures holds that in describing a data structure we should give a set of behavioral laws including perhaps a set of growth or combining laws in addition to or instead of a description of static structure. This view is quite reasonable. Perhaps the way to resolve it with the previous view is to introduce a terminological distinction in which a data structure is defined as consisting of a class of static structures, and a data type is defined as consisting of a class of (behaviorally equivalent) data structures together with a set of behavioral laws that they must obey. For example, stacks, dates
and matrices would be data types whereas directed graphs and
arrays would be data structures.

These observations serve to emphasize that this thesis
offers but one approach to the study of data structures. Others
are possible and can and should be explored. Thus, in conclusion,
it is probable that in studying data structures this thesis has
merely scratched the surface, but we hope it has scratched
provocatively.
APPENDIX I

Additions to the Revised Algol Report Defining Syntax

Note: The notation \(< x > ::= \ldots < y >\) denotes that the construction \(< y >\) is to be added to the definition of the construction \(< x >\) in the Revised Algol Report. For example, \(<\text{specificator} > ::= \ldots \text{descriptor}\) denotes the addition of the construction descriptor to the syntax equation \(<\text{specificator} > ::= \text{string} | \text{value} | \text{label} | \text{descriptor}\). Sections of the Revised Algol Report in which the additions are to take place are denoted by the section numbers of the report.

2.3 Delimiters

\(<\text{declarator} > ::= \ldots <\text{descriptor variable} > | <\text{descriptor function designator} > | \text{descriptor} | \text{path} | \text{reference} | \text{any} | \text{identifier}\)

\(<\text{descriptor operator} > ::= <\text{the mark }"\text{"} > | : | :: | \lor | \land |
\equiv | \otimes | \Delta | \nabla\)

\(<\operator > ::= \ldots <\text{descriptor operator} >\)

3. Expressions

\(<\text{expressions} > ::= \ldots <\lambda\text{-expression} > | <\text{descriptor expression} > | <\text{constructor expression} > | <\text{selection expression} > | <\text{reference expression} > | <\text{path expression} >\)

3.2 Function Designators

\(<\text{compound function designator} > ::= <\text{function designator} >
<\text{actual parameter part} >\)

\(<\text{dotted function designator} > ::= <\text{procedure identifier} >.
<\text{actual parameter part} >\)

\(<\text{function designator} > ::= \ldots <\text{dotted function designator} > | <\text{compound function designator} >\)
Examples
List.(complex)
Rowarray(m, complex)(X)

3.2.4 Standard Functions

\textbf{type}(E, D) \quad E \text{ is a data valued expression, } D \text{ is a descriptor valued expression, and the value of type}(E, D) \text{ is an integer.}

\textbf{path}(d, D) \quad \text{Path is a path valued function giving the path to the first instance of the subdescriptor } d \text{ in the descriptor } D, \text{ where } d \text{ and } D \text{ are descriptor expression.}

\textbf{copy}(E) \quad \text{Copies the top format of } E. \ E \text{ is a data valued expression.}

\textbf{copyall}(E) \quad \text{Copies all formats and subformats of } E \text{ recursively. } E \text{ is a data valued expression.}

3.4 Boolean Expressions

\begin{align*}
\langle \text{relation} \rangle : &= \ldots \langle \text{expression} \rangle = \langle \text{descriptor expression} \rangle | \\
&\quad \langle \text{expression} \rangle = \langle \text{expression} \rangle
\end{align*}

\ldots \text{ Add six sections to the definition of Expressions as follows:}

3.6 \text{ } \lambda \text{- Expressions}

\begin{align*}
\langle \lambda \text{- expression} \rangle &::= \{ \lambda \left( \langle \text{bound variable list} \rangle \right) \langle \text{expression} \rangle \} \\
\langle \text{bound variable list} \rangle &::= \langle \text{identifier} \rangle | \langle \text{bound variable list}, \langle \text{identifier} \rangle \rangle \\
\langle \text{monadic, Boolean } \lambda \text{-expression} \rangle &::= \{ \lambda (\langle \text{identifier} \rangle) \langle \text{Boolean expression} \rangle \}
\end{align*}

Examples
\begin{align*}
\{ \lambda(x, y) \ C(\text{map}(x), y) \} \\
\{ \lambda(y) \ B(y) \land (C(y) \lor K(y)) \}
\end{align*}
3.7 Descriptor Expressions

3.7.1 Dotted Descriptors and Descriptor Names

\[
\begin{align*}
< \text{descriptor name} > &::= < \text{descriptor identifier} > \mid \\
& \quad < \text{descriptor function designator} > \\
< \text{dotted descriptor} > &::= < \text{dotted descriptor identifier} > \mid \\
& \quad < \text{dotted descriptor function designator} >
\end{align*}
\]

Examples
List, (complex)
BinaryFormula
UnaryFormula

3.7.2 Component Descriptions

\[
\begin{align*}
< \text{augmented type} > &::= < \text{type} > \mid \text{any} \mid \text{identifier} \mid \text{path} \mid \text{reference} \\
< \text{data space specifier} > &::= < \text{augmented type} > \mid < \text{dotted descriptor} > \mid \\
& \quad < \text{string} > \mid = < \text{expression} > \mid < \text{descriptor expression} > \mid \\
& \quad < \text{monadic, Boolean } \lambda \text{-expression} > \mid \text{nil} \\
< \text{component description} > &::= < \text{data space specifier} > \mid \\
& \quad < \text{identifier} > : < \text{data space specifier} >
\end{align*}
\]

Examples
Data Space Specifiers
\[
\begin{align*}
\text{real} \\
\text{List, (integer)} \\
\text{'+ '} \\
= (N + 1) \\
\{ \lambda(x) \text{ integer}(x) \land x > 0 \} \\
\text{nil}
\end{align*}
\]

Component Descriptions
\[
\begin{align*}
\text{realpart : real} \\
\text{coordinates : [ x: real | y: real ]}
\end{align*}
\]
3.7.3 Elementary Descriptors

\[
\langle \text{component sequence} \rangle ::= \langle \text{component description} \rangle \mid \\
\langle \text{component sequence} \rangle \langle \text{the mark "|"} \rangle \langle \text{component description} \rangle \\
\langle \text{replicated descriptor} \rangle ::= \langle \text{arithmetic expression} \rangle \times [\langle \text{component description} \rangle] \\
| \langle \text{variable} \rangle \rightarrow \langle \text{arithmetic expression} \rangle \times [\langle \text{component description} \rangle] \\
| \langle \text{indefinite} \rangle \times [\langle \text{component description} \rangle] \\
\langle \text{tagged descriptor} \rangle ::= \langle \text{descriptor name} \rangle \\
\langle \text{elementary descriptor} \rangle ::= [\langle \text{component sequence} \rangle] \\
| \langle \text{descriptor name} \rangle ::= [\langle \text{component sequence} \rangle] \\
| \langle \text{replicated descriptor} \rangle \mid \langle \text{tagged descriptor} \rangle
\]

Examples

Component Sequences

real | \{\lambda (x) \ B(x)\} \mid \text{Boolean} \\
locus : [\ x: \text{integer} \mid y: \text{integer} \mid \text{mark} : \text{Boolean}]

Replicated Descriptors

i \rightarrow (N + 1) \times \[\text{string}\] \\
\text{indefinite} \times [\text{real}] \\
N \times [\ t ]

Tagged Descriptors

" BinaryFormula

Elementary Descriptors

\text{Triple} :: \{\ \text{datum: any} \mid 11 : .D \mid r1 : .D\} \\
\text{Pair} (t) :: [\ \text{Head} : t \mid \text{Tail} : \text{List} .(t) \]
3.7.4 Modified Descriptors

< format modifier > ::= < format modifier conjunct > |
    < format modifier > ^ < format modifier conjunct >
< format modifier conjunct > ::= ( < component designator > =
    < data space specifier > )
< component designator > ::= < path expression > | < identifier >
< format modified descriptor > ::= < elementary descriptor > @
    < format modifier >
< predicate modified descriptor > ::= < descriptor expression > ∃
    < λ-expression >
< constructor modified descriptor > ::= < descriptor expression > Δ
    < λ-expression >
< modified descriptor > ::= < format modified descriptor > |
    < predicate modified descriptor > |
    < constructor modified descriptor >

Examples

Format Modified Descriptor

BinaryFormula Θ (LeftOperand = List.(identifier)) ∧
    (Operator = (= 'λ'))

Predicate Modified Descriptor

SymmetricListElement ⊃ { λ(x) ( ll(x) = nil v rl(ll(x))=x ) ∧
    (rl(x) = nil v ll(rl(x))=x ) }

Constructor Modified Descriptor

SymmetricListElement Δ { λ(x) if ~Nullist(x) then ll(r1(x)) ← x }
3.7.5 Descriptor Expressions

\[
\langle \text{character list} \rangle ::= \langle \text{string with no commas} \rangle | \\
\langle \text{character list} \rangle , \langle \text{string with no commas} \rangle \\
\langle \text{character set} \rangle ::= \{ \langle \text{character list} \rangle \} \\
\langle \text{descriptor expression} \rangle ::= \langle \text{elementary descriptor} \rangle | \\
\langle \text{descriptor name} \rangle | \text{declarand} | \langle \text{character set} \rangle | \\
\langle \text{descriptor expression} \rangle \lor \langle \text{descriptor expression} \rangle | \\
\langle \text{descriptor expression} \rangle \land \langle \text{descriptor expression} \rangle
\]

Examples

Character Sets
\
{+} \\
{north, east, south, west}

Descriptor Expressions

\[
\text{Nullist} ::= [\text{nil}] \lor \text{Pair}(t) ::= [\text{Heat}: t | \text{Tail}: \text{List}:(t)]
\]

\text{declarand}

\[
[\text{real} | \text{real} | \text{real}] \land [\text{integer} | \text{integer}]
\]

3.8 Constructor Expressions

3.8.1 Possibly Diminished Actual Parameter Parts

\[
\langle \text{possibly diminished actual parameter} \rangle ::= \langle \text{empty} \rangle | \\
\langle \text{actual parameter} \rangle | \langle \text{possibly diminished actual parameter part} \rangle \\
\langle \text{possibly diminished actual parameter list} \rangle ::= \\
\langle \text{possibly diminished actual parameter} \rangle | \\
\langle \text{possibly diminished actual parameter list} \rangle , \\
\langle \text{possibly diminished actual parameter list} \rangle \\
\langle \text{possibly diminished actual parameter part} \rangle ::= \langle \text{empty} \rangle \\
(\langle \text{possibly diminished actual parameter list} \rangle)
\]
3.8.2 Constructor Expressions

< constructor expression > ::= 
   cons (<descriptor expression>) <possibly diminished actual parameter part>

Examples

Constructor Expressions
cons(D)
cons(D) (a, b, , )
cons(d) (a, b, (c, d, , e),)
cons(d) (a, b, (c, d, f, e), g)

3.9 Selection Expressions

< selection expression > ::= < function designator > | < subscripted variable> |
                           < subscripted function designator >

< subscripted function designator > ::= < function designator><subscriptlist> |
                                     < subscripted function designator > [ <path expression>]

Examples
index( rowmatrix[5])
base (x)
matrix ( numerator ( X[2]))[3, 5]
valence ( bondlist (x) [4] )
M [2][4][6]
3.10 Reference Expressions

< pointer expression > ::= → < selection expression >
< contents expression > ::= = the mark "<" < expression > < the mark ">"
< reference expression > ::= < pointer expression > | < contents expression >

Examples

→ valence ( bondlist(x) [4])
<< D[3, 2] >>

3.11 Path Expressions

< path expression > ::= < path variable > | < path function designator > |
< subscript list > | [< integer valued expression >]
< path expression > ⊗ < path expression >

Examples

[ 1, 2, (N + 1) ]
p ⊗ [2]
sidpath(x, y) ⊗ [2] ⊗ p

4.2 Assignment Statements

< left part > ::= ... initial value ← | result ← | < selection expression > ← |
< selection expression > ←← | < variable > ←← |
< procedure identifier > ←←

< assignment statement > ::= ... < left part list > < expression >

5.1 Type Declarations

< type > ::= ... < descriptor name > | descriptor | identifier | path | any | reference | string
5.4 Procedure Declarations

Note that we can have descriptor, path, reference, etc. procedures since in

< procedure declaration > ::= . . . < type > procedure
< procedure heading > < procedure body >
the '<type>' can be descriptor, path, reference, etc.

Example

descriptor procedure List(t); descriptor t;
List ← Nullist:|[nil] ∨ Pair(t):|[head:t Tail:List.(t)] ;
APPENDIX II

Summary of Semantics

1. Component descriptions describe the data spaces from which components of data formats may be chosen. The data space specifier of a component description is either a type word, a unit data space described by an expression preceded by a unary =, a monadic Boolean lambda expression, a quoted character string, or a descriptor valued expression describing the structure of a nested format.

2. If a component description bears a name this name may be used as the name of the selector function which accesses the component from a datum constructed according to the format given by the descriptor in which the component description occurs.

3. Elementary descriptors are compositions of component descriptions which compositions include (a) composition into linear formats by use of the operator "|", (b) nesting of descriptors, (c) recursive composition in which the name of the format being defined is a component of the composed format, and (d) iterative composition in which a component description is iterated n times possibly subject to variation depending on successive values of a unit stepping controlled variable or iterated an indefinite number of times, which number is determined at allocate or construct time depending on the number of parameters supplied to the constructor function. An elementary descriptor may also be a tagged descriptor variable or descriptor function designator causing its associated constructors and predicates to construct and to test for tagged data. Linear compositions of descriptors may be concatenated.

4. Elementary descriptors may bear names, which names, in turn, may bear parameters, and may be used in constructors, predicates and declarations.

5. Modified descriptors are of three sorts: (a) format modified descriptors, (b) predicate modified descriptors, and (c) constructor modified descriptors. Each modified descriptor consists of a modificand, which is an elementary descriptor or descriptor valued expression, and a modifier. A format modifier is used to alter, to expand, or to restrict the format described by its modificand. Components to be altered or added in a format are specified either by name or by subscript list. Predicate and constructor modifiers provide extensions.
in the form of lambda expressions to predicates and constructors respectively that are associated with the modificand.

6. Descriptor expressions are either elementary descriptors, modified descriptors or disjunctions of elementary and modified descriptors, which disjunctions describe the alternate data formats that a datum may possess. In the case of disjunctions whose disjuncts describe strings, a character set abbreviation may be used to provide a more concise but equivalent descriptor.

7. Descriptor formulae may be computed as well as written. Variables and function designators which are to take on descriptors as values must be declared of type descriptor. In computing descriptors both the format composition operator "\[\]
" and the format concatenation operator "\(\bigcirc\)" may be used to combine parts of formats in nested or linear fashion. The symbol nil functions as an identity element under these forms of composition of formats.

8. The function cons applied to descriptor valued expressions produces a constructor function which constructs from lists of parameters data of the format described by the value of the descriptor expression. The lists of parameters may be incomplete, in which case space will be allocated for data components corresponding to missing parameters and the values of these components will be left undefined. When cons is applied to a descriptor which is a disjunction, a representation of the first disjunct is constructed. An initial value policy permits constructed data to assume arbitrary initial values. When cons is applied to an indefinite replicator the number of replications of components is determined by the length of the parameter list to which the corresponding constructor function is applied. Construction of tagged data is denoted by applying cons to tagged descriptors. Modified construction is evoked by applying cons to constructor modified descriptors, which modification is performed by applying the lambda expression in the modifier to the result of constructing the format given by the modificand.

9. Either the name of a component or its ordinal position within a given format may be used in selection expressions to select the component from a datum constructed according to the given format. If components are not named then integer selection must be used. In selection expressions which combine component name selection and integer selection, component name selection takes precedence. In selection expressions, integer selectors may be given by subscript
lists, by values of path expressions, or by repeatedly appending expressions of the form \([\text{integer}]\) to the right hand side of a selection expression.

10. Selection expressions may be used both on the left and on the right hand sides of assignment statements. On the right they produce selected data values; on the left they signify a location in an allocated datum where assignment of a datum is to take place.

11. Descriptor formulae and names of elementary descriptors may be used in the construction \(<\text{expression}> == D\) to test whether the datum that is the value of the expression is an instance of D. If D is a tagged descriptor of the form "d then \(<\text{expression}> == "d\) tests to see if the datum given by the value of the expression bears the tag d. If D is a predicate modified descriptor then the predicate \(<\text{expression}> == D\) is true if and only if the Boolean lambda expression in the predicate modifier yields the value true when applied to the datum which is the value of the expression. Functional notation D(x) may be employed as a synonym for the predicate notation x == D.

12. The function type(x, D) with x a data valued expression and D a descriptor valued expression gives as value an integer which is the ordinal position of the first disjunct in D of which x is an instance, if any. If no such disjunct exists the value is 0. Thus even though alternate disjuncts are not named in a descriptor their types may be referenced.

13. x=y is a recursive equality predicate among constructed data which tests whether x and y have the same format and equal component values. The predicate is well defined if x and y do not contain loops.

14. In descriptor procedures the construction \(\text{initial value } \gets <\text{expression}>\) may be used to specify an initial value for a variable which is declared using the function designator. If the \(<\text{expression}>\) is 'declarand' then the name of the variable being declared becomes the initial value. Descriptor valued variables, descriptor valued function designators and elementary descriptor names may be used as declarators. The declarator any may be used to declare variables and procedures whose values may be of any type.

15. Pointer expressions, which consist of selection expressions preceded by the unary operator "\(-\)", have as values the left hand values of the
the selection expressions. These values are of type reference. Variables or procedures may be of type reference and may take on reference values. Reference values may be used in the construction of data structures. The contents operation \(< r >\) obtains the right hand value corresponding to the left hand value \(r\).

16. The overlay assignment of the form \(e_1 \leftarrow e_2\) moves the datum specified by the right hand value of \(e_2\) so that it occupies the location specified by the left hand value of \(e_1\). After execution, reference to the right hand value of \(e_1\) gives the right hand value of \(e_2\). The function copy(e) produces a copy of the structure allocated for the linear format of the right hand value of \(e\) but copies no nested subformats, whereas copyall(e) produces a copy of the structure allocated for all formats and nested subformats of the right hand value of \(e\). In the latter case the right hand value of \(e\) must involve no self referential loops.

17. Variables and function designators declared of type path may take on access paths as values. These access paths are subscript lists and may be computed and concatenated. Access paths may be applied to a datum in a selection expression to produce as a result of a succession of or dinal selections access to components of nested formats. If \(d\) is a subexpression of the descriptor \(D\) then path(d, D) has as value the access path to the first instance of \(d\) in \(D\) under the assumption that search is performed in sequence element mode.

18. Parallel assignments and block expressions are permitted in the composition of programs.
APPENDIX III

Data Structure Definitions

1. Formulae

descriptor Atom, Formula, UnaryFormula, BinaryFormula,
UnaryOperator, BinaryOperator, ArrayFormula, ProcedureFormula,
LambdaFormula, ProcedureOperator, PatternPrimary;
Atom ← [real] ∨ [integer] ∨ [Boolean] ∨ [string] ∨ [identifier] ∨
PatternPrimary;
UnaryFormula ← [Operator: UnaryOperator | Operand:Formula ];
BinaryFormula ← [Operator: BinaryOperator | LeftOperand:Formula | RightOperand:Formula];
Formula ← UnaryFormula ∨ BinaryFormula ∨ Atom ∨ List.(Formula) ∨ LambdaFormula ∨ ProcedureFormula ∨ ArrayFormula;
UnaryOperator ← {-, _,., sin, cos, tan, ln, exp, arctan, =, ==, ~, of};
BinaryOperator ← {+, -, ×, /, ↑, :, =, =, =, [., (, ∨, ^, ¬, <, >, λ} ∨ [=!, !];
PatternPrimary ← {real, integer, Boolean, formula, atom, list, any};
ArrayFormula ← "BinaryFormula Θ (Operator ={[}∧(LeftOperand =identifier)
(RightOperand = List(Formula)));
ProcedureFormula ← "BinaryFormula Θ (Operator ={[]}∧
(LeftOperand = ProcedureOperator) ∧ (RightOperand = List(Formula)));
LambdaFormula ← "BinaryFormula Θ (Operator ={λ} ∧
(LeftOperand = List(identifier)));
ProcedureOperator ← identifier ∨ LambdaFormula ∨ ArrayFormula ∨
UnaryOperator ∨ BinaryOperator;

descriptor procedure List(t); descriptor t;
List ← Nullist ::[nil] ∨ Pair(t) ::["BinaryFormula Θ
(Operator = (=!, !))∧(LeftOperand=t)∧(RightOperand=List(t))];
2. Lists

a. One Way Lists

\texttt{descriptor atom, element, List;}
\texttt{element \leftarrow atom \lor List;}
\texttt{List \leftarrow \text{Nullist}:[\text{nil}] \lor \text{Pair}:[\text{Head}:element|\text{Tail}:.List];}
\texttt{atom \leftarrow \text{identifier};}

b. One Way Lists - Type Dependent

\texttt{descriptor procedure List(t); descriptor t;}
\texttt{List \leftarrow \text{Nullist}:[\text{nil}] \lor \text{Pair}:[\text{Head}:t|\text{Tail}:\text{List}.(t)];}

c. Symmetric Lists

\texttt{SymmetricListElement \leftarrow}
\texttt{Nullist}:[\text{nil}] \lor \texttt{Triple}:[[\text{datum}: \text{any}|\text{ll}:.\text{SymmetricListElement}|\text{rl}:.\text{SymmetricListElement}]
\texttt{\Delta \{\lambda (x) \text{ if } \sim\text{Nullist}(\text{rl}(x)) \text{ then } \text{ll}(\text{rl}(x)) \leftarrow x\} \}
\texttt{\exists \{\lambda (y) (\text{Nullist}(\text{ll}(y)) \lor \text{rl}(\text{ll}(y))=y) \land (\text{Nullist}(\text{rl}(y)) \lor \text{ll}(\text{rl}(y))=y)\} \}

d. Threaded Lists

\texttt{descriptor Header, DataElement, Terminator, TLE;}
\texttt{Header \leftarrow \text{top}:[[\text{pointer}:\text{TLE}] \lor \text{nontop}:[[\text{pointer}:\text{TLE}|\text{link}:\text{TLE}];}
\texttt{DataElement \leftarrow [[\text{datum}: \text{any}|\text{link}: \text{TLE}];}
\texttt{Terminator \leftarrow [[\text{link}: \text{TLE}];}
\texttt{TLE \leftarrow \text{Header} \lor \text{DataElement} \lor \text{Terminator};}

e. Rings

\texttt{descriptor pair;}
\texttt{pair \leftarrow [[\text{datum}: \text{any}|\text{link}: \text{.pair}];}
\texttt{descriptor procedure Ring(n); integer n;}
\texttt{Ring \leftarrow (n \times \text{pair}) \Delta \{\lambda (x) \text{ LinkRing}(n,x)\} \}
procedure LinkRing(n, x) ; integer n ; Ring(n) x ;
begin integer i ;
  for i ← 1 step 1 until n-1 do link(x[i]) ← x[i+1] ;
  link(x[n]) ← x[1] ;
end ;

3. Algol Text

descriptor conditional, goto, assignment, declaration, statement,
labelledstatement, block ;
conditional ← [condition:Formula | truearm: any | falsearm: any ] ;
goto ← [label: identifier ] ;
assignment ← [ lhs: Formula | rhs: Formula ] ;
declaration ← [ declarator:Formula|declarandlist: List(identifier) ] ;
labelledstatement ← [label: identifier | text : statement ] ;
block ← [ head: List(declaration)|body: List(statement) ] ;
statement ← conditional ∨ goto ∨ assignment ∨ labelledstatement ∨
  block ∨ dummy::[nil] ;

4. Flow Charts

descriptor flowchart, actionbox, decisionbox ;
actionbox ← [body:Formula | exit: flowchart | mark: Boolean ]
decisionbox ← [test: Formula | trueexit: flowchart|falseexit: flowchart];
flowchart ← actionbox ∨ decisionbox ∨ nullbox :: [nil] ;

5. Electronic Circuits

descriptor Resistor, Capacitor, Inductor, Part, Circuit ;
Resistor ← [resistance: Formula ] ;
Capacitor ← [capacitance : Formula ] ;
Inductor ← [ inductance : Formula ] ;
Part ← Resistor ∨ Capacitor ∨ Inductor ;
Circuit ← Part ∨ SeriesElement::[Circuit | . Circuit] ∨
  ParallelElement::[. Circuit | . Circuit] ;
6. Complex Variables

descriptor complex
complex ← [realpart:real | imagpart:real ];

7. Files

a. Organic Compound Files
Record ← [Number: integer | Name: string | Synonyms: string |
Formula: string | MolWt: real | Properties: string |
Density: Formula | MeltingPt: Formula | BoilingPt: integer |
Solubility: [inWater:Formula | inAlcohol:Formula | inOther: string ] ];
OrganicCompoundFile ← N × [Record];

b. Medical Files
descriptor MedicalFile, medicinespecifier, dietcode ;
MedicalFile ← [patientname: string | roomnumber: integer |
doctor: string | admissiondate : List(integer) |
medicationschedule : List(medicinespecifier) | diet: dietcode ];
medicinespecifier ← [medicinename: identifier | dosage : Formula |
schedule : List(string) ];
dietcode ← softsolid, liquid, intravenous, normal ;

c. Library Cards
descriptor AuthorCard, SubjectCard, TitleCard, Book ;
Book ← [Title: string | Author: string | Publisher: string | pages: integer |
size: Formula | date: Formula | precis: List(string) ];
AuthorCard ← [ Author : string | referent : Book ];
SubjectCard ← [ Subject : string | referent : Book ];
TitleCard ← [ Title : string | referent : Book ];
d. Abstract Files of Predetermined Size

descriptor procedure Field(n); integer n; Field ← n × [bit];

descriptor procedure CharSeq(n); integer n; CharSeq ← n×[character];

descriptor letter, digit, bit, character;

bit ← {0, 1}; digit ← {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

letter ← {a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z};

character ← letter ∪ digit;

descriptor procedure UnsignedInt(n); integer n; UnsignedInt ← n×[digit];

descriptor procedure SubRecord(n); integer n;

SubRecord ← Field(n) ∪ CharSeq(n) ∪ UnsignedInt(n);

descriptor procedure Record(m, n); integer m, n;

Record ← m × [ SubRecord(n) ];

descriptor procedure File(k, m, n); integer k, m, n;

File ← k × [Record(m, n) ];

e. Abstract Files of Arbitrary Size

descriptor File, Record, Subrecord;

SubRecord ← indefinite × [ character ];

Record ← indefinite × [ SubRecord ];

File ← indefinite × [Record];

8. Matrices

descriptor procedure Matrix(m, n, t); integer m, n; descriptor t;

Matrix ← m × [ n × [ t ] ];

descriptor procedure LowerTriangularArray(m, t); integer m;

LowerTriangularArray ←(i → m × [ i × [ t ] ]);
9. Reduction Recognizers for Graphical Data

descriptor graph, realpair, node, edge, subscriptnode, designator,
metacharacter, reductiongraph, ReductionSet, Reduction;

graph ← [ locus: realpair | nodelist: List(node) | edgelist :List(edge) ];
realpair ← [ x:real|y:real ];
node ← [ name: identifier | x:real|y:real|semantics: any | mark: Boolean ];
edge ← [ x:real| y:real | deltax: real | deltay: real | color: identifier |
mark: Boolean | correspondent: edge ];
subscriptnode ← [ name: designator | subscript: integer | x:real|y:real |
correspondent: node ];
designator ← metacharacter \[identifier\] \[sigma\] ;
metacharacter ← List(identifier); 
reductiongraph ← [locus: realpair | nodelist : List(subscriptnode) |
edgelist : List(edge) ];
ReductionSet ← List(Reduction); 
Reduction ← [ label : identifier | LeftGraph : reductiongraph |
arrow: Boolean | RightGraph : reductiongraph | action : Formula |
link : identifier ];

10. Organic Chemistry Molecules

descriptor procedure Atom(N, I); integer N; identifier I;

Atom ← [ name: (=I) | valence: (=N) | bondlist: N × any |
mark : Boolean ];

descriptor carbon, hydrogen;

carbon ← Atom(4,.C);
hydrogen ← Atom(1,.H);
11. Descriptors

Type ← {real, Boolean, integer, string, any, reference, path, descriptor identifier};
DataSpaceSpecifier ← Type \( \land \) UnitDataSpace \( \land \) [string] \( \land \) ProcedureFormula \( \land \) [nil] \( \land \) Descriptor;
UnitDataSpace ← 'UnaryFormula \( \land \) (Operator = \{=\})
ComponentDescription ← DataSpaceSpecifier \( \land \) 'BinaryFormula \( \land \) (Operator = \{:\}) \land \) (LeftOperand = identifier) \( \land \) (RightOperand = DataSpaceSpecifier);
ElementaryDescriptor ← ReplicatedDescriptor \( \land \) TaggedDescriptor \( \land \)
'BinaryFormula \( \land \) (Operator = \{=\}) \land \) (LeftOperand = DescriptorName) \land \) (RightOperand = . ElementaryDescriptor) \land \) indefinite \( \times \) ComponentDescription;
TaggedDescriptor ← 'UnaryFormula \( \land \) (Operator = \{tag\}) \land \) (Operand = DescriptorName);
DescriptorName ← identifier \( \land \) ProcedureFormula;
ReplicatedDescriptor ← [replicator: [cv:Formula | limit:Formula] \( \land \) {indefinite}] \land \) replicand : ComponentSpecifier;
FormatModifiedDescriptor ← 'BinaryFormula \( \land \) (Operator = \{\theta\}) \land \) (LeftOperand = ElementaryDescriptor) \land \) (RightOperand = FormatModifier);
FormatModifier ← 'BinaryFormula \( \land \) (Operand = \{\theta\}) \land \) (LeftOperand = FormatModifierConjunct) \land \) (RightOperand = . FormatModifier) \land \) [nil];
FormatModifierConjunct ← 'BinaryFormula \( \land \) (Operator = \{\theta\}) \land \) (LeftOperand = ComponentSpecifier) \land \) (RightOperand = DataSpaceSpecifier);
ComponentSpecifier ← identifier \( \land \) List(integer);
ConstructorModifiedDescriptor ← 'BinaryFormula \( \land \) (Operator = \{\Delta\}) \land \) (LeftOperand = . Descriptor) \land \) (RightOperand = LambdaFormula);
PredicateModifiedDescriptor ← 'BinaryFormula \( \land \) (Operator = \{\varepsilon\}) \land \) (LeftOperand = . Descriptor) \land \) (RightOperand = . LambdaFormula);
ModifiedDescriptor ← FormatModifiedDescriptor \( \land \) ConstructorModifiedDescriptor \land \) PredicateModifiedDescriptor;
CharacterSet ← List(string);
Descriptor ← ElementaryDescriptor \( \land \) ModifiedDescriptor \( \land \) {declarand} \land \) CharacterSet \( \land \) 'BinaryFormula \( \land \) (Operator = \{\varepsilon\}) \land \) (LeftOperand = . Descriptor) \land \) (RightOperand = . Descriptor);
REFERENCES


23. Floyd, R.W., Special Classes of Context-Free Languages and Their Recognizers, NATO Summer School in Programming Languages, Villard de Lans, (Sept. 1966)


41. Moses, J., Symbolic Integration, Artificial Intelligence Project Memorandum No. 97, MIT, Project MAC, (Jun. 1966)


51. Quatse, Jesse T., A Visual Display System Suitable for Time Shared Use, internal publication, Carnegie Institute of Technology, Pittsburgh, (June 1965)

52. Roberts, Lawrence G., Graphical Communication and Control Languages, Second Congress on Information System Sciences, Hot Springs, Virginia, (1964)


67. Weizenbaum, J., Symmetric List Processor, CACM, vol. 6, no. 9, (Sept. 1966), pp. 524-543

68. Wells, CACM, (Nov. 1963)
This dissertation presents a descriptive notation for data structures which is embedded in a programming language in such a way that the resulting language behaves as a synthetic tool for describing data and processes in a number of application areas. A series of examples including formulae, lists, flow charts, Algol text, files, matrices, organic molecules and complex variables is presented to explore the use of this tool. In addition, a small formal treatment is given dealing with the equivalence of evaluators and their data structures.
<table>
<thead>
<tr>
<th>KEY WORDS</th>
<th>LINK A</th>
<th>LINK B</th>
<th>LINK C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ROLE</td>
<td>WT</td>
<td>ROLE</td>
</tr>
</tbody>
</table>

Security Classification