

INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.
2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.
3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of "sectioning" the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.
4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.
5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.

**University
Microfilms
International**
300 N. Zeeb Road
Ann Arbor, MI 48106

8306724

Mejstrik, Norman Louis

**FORMAL SPECIFICATION OF OPERATIONS ON A CLASS OF
SYNTACTICALLY-SPECIFIED DATA STRUCTURES**

The University of Oklahoma

PH.D. 1982

**University
Microfilms
International** 300 N. Zeeb Road, Ann Arbor, MI 48106

Copyright 1982

by

Mejstrik, Norman Louis

All Rights Reserved

THE UNIVERSITY OF OKLAHOMA
GRADUATE COLLEGE

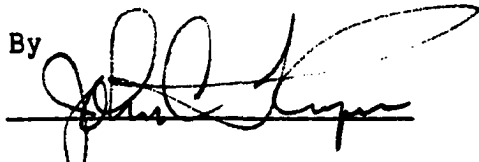
FORMAL SPECIFICATION OF OPERATIONS
ON A CLASS
OF SYNTACTICALLY-SPECIFIED DATA STRUCTURES

A DISSERTATION
SUBMITTED TO THE GRADUATE FACULTY
in partial fulfillment of the requirements for the
degree of
DOCTOR OF PHILOSOPHY

By
NORMAN LOUIS MEJSTRIK
Norman, Oklahoma
1982

FORMAL SPECIFICATION OF OPERATIONS
ON A CLASS
OF SYNTACTICALLY-SPECIFIED DATA STRUCTURES
A DISSERTATION APPROVED FOR
THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

By



Bobbie Foote

Billy K. Walker

John T. Minor

Al. R. Hersen

© by Norman L. Mejsrik 1982
All Rights Reserved

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude and appreciation to my major professor, Dr. John C. Thompson, for his ideas and patient guidance and counsel during my dissertation work. I also thank the members of my committee, Dr. Foote, Dr. Hurson, Dr. Minor and Dr. Walker, for their helpful comments.

I am deeply indebted to Mr. Q. Inks Franklin for his advice and generous computer support during the production of this dissertation.

I am very grateful to my wife, Carolyn, and daughter, Marie, for their patience and understanding during my graduate years. Finally, I thank my parents, Ernest and Agnes, for their continual encouragement.

CONTENTS

ACKNOWLEDGEMENTS	iv
CONTENTS	v
LIST OF FIGURES	vii
1. Introduction	1
1.1 Objective	1
1.2 Motivation	1
1.3 Previous Work	3
1.4 Introduction to Formal Data Structure Specification	6
2. Underlying Theory	8
2.1 Introduction	8
2.2 Multirelational Graphs and K-formulas	9
2.3 Grammars and Graph Grammars	11
2.4 Grammars Augmented for Node Identification	14
2.5 Context-Free Languages and Recognizers	17
2.6 Recognizable Structures	22
3. Formalized Data Structure Operations	39
3.1 K-Grammars for Several Common Data Structures	39
3.2 The Correspondence of K-Grammars and Classical Data Structure Generation	43
3.3 Operations on Data Structures	49
3.4 Formalisms for Operations on Data Structures	55
4. Application of Data Structure Transforms	64
4.1 Method of Application	64
4.2 Syntax Checking Transform Statements	65
4.3 Interpreter Operations	67
4.4 Examples of Interpreter Use	71
5. Summary, Future Work and an Evaluation	86
5.1 Summary	86
5.2 Future Work	86
5.3 An Evaluation	88

BIBLIOGRAPHY	91
APPENDIX 1: Interpreter Documentation .	93
APPENDIX 2: "lex" Specification . . .	109
APPENDIX 3: "yacc" Specification . . .	110

LIST OF FIGURES

<u>Figure</u>	<u>Title</u>	
3.2.1	Binary Tree Constructor Algorithm	44
3.3.1	Derivation Trees for Singly-Linked List Insertion .	52
4.3.1	Interpreter Data Flow	69
4.3.2	Interpreter Software Structure	70
4.4.1	Message Queue Example	74
4.4.2	AVL Tree "LL" Rotation	71
4.4.3	Binary Tree Rebalancing	78
4.4.4	AVL Tree "LL" Rotation Example	81
4.4.5	AVL Tree "LR" Rotation Example	83

CHAPTER ONE

1. Introduction.

1.1 Objective.

This paper presents a method of formally specifying operations on a certain class of linked data structures. Rooted in formal language theory, the specification of operations provides a basis for demonstrating the correctness of the operations, and a vehicle for future implementation of a mechanized programming system for data structures of interest.

1.2 Motivation.

Linked data structures form an important part of contemporary computing technology. Examples are found in operating system scheduler queues, file system directory trees, hierarchical and network data base management system implementations, and first-in-first-out message switching systems. Fundamentals of linked data structures are taught in undergraduate data structures courses, since an understanding of these structures is prerequisite to comprehending many other aspects of computer science coursework.

In computer science classrooms and in technical journals alike, the usual method of describing linked data

structures involves diagrams showing the data elements and their relation to one another. The data elements are the nodes of a directed graph, and the relationships between elements are expressed as edges between the nodes. Once the intuitive graphic description has been communicated, authors frequently use programming language-like procedural descriptions of operations on the structures. Such algorithmic descriptions inherently include considerable implementation detail which is of secondary interest when describing structure transformations. While the graphical depiction of structure operations has a good deal of intuitive appeal, to rely on graphical methods alone results in a lack of desired conciseness and precision.

Various approaches to formalizing the description of data structure operations are addressed in the next section. The application of formalism results in a standard method of communicating ideas on the subject involved. Formalism provides the opportunity to analyze implementations of data structure operations at the abstract level, before in situ structures and programming mechanisms are employed. Suitable choice of notation allows conventional processing by computer, contrasted with the need for graphical computer input and output devices when relying solely on graphical depiction. The conventional processing feature also facilitates automated support of proofs of the correctness of data structure operations. Achieving the goal of

elegance of formalism means achieving conciseness and precision without the complexity generally found in a programming language approach. This paper addresses the goal of an elegant method of specifying operations on linked data structures.

1.3 Previous Work.

In the past, the most frequently used method of defining linked data structures and describing operations on them has been the use of a programming language. The primary programming language features used are arrays, pointers, and the instructions (statements) which operate on them. These conventional features are used to synthesize data structures and operations on them because the structures themselves are not inherent in general purpose programming languages. When directed graphs are used to represent linked data structures, programming language extensions such as those described by Crespi-Rehizzi and Marpurgo (CM70) become available to implement data structure operations which are analogous to graph union, intersection and subtraction. Schneiderman and Scheuermann (SS74) also proposed an extension to a host programming language to include linear structure and multistrukture declarations, and facilities for operations on these structures. However, because of the large amount of detail required, the programming language extension method by itself contributes little to the desired goal of demonstrating the correctness

of the operations performed.

Earley (EJ71) attacked the data structure problem more directly through the use of so-called "V-graphs." Significant in the Earley paper is the distinction between data structure semantics (how the data is stored and accessed, and how the structure may be changed) and implementation (how semantics are realized in a physical machine). This important distinction, which is evident in other works on both graphs and data structures (GY75, RA71, PF71), allows one to model the real world problem with semantics without extraneous implementation detail. The implementation problem is attacked only after the semantics of the logical model of the problem are clearly understood. This formal recognition of the dichotomy between abstraction and implementation is essential to practical resolution of the correctness issue.

In addition to the programming language extension and directed graph approaches to data structures, Horowitz and Sahni (HS76) and Guttag (GJ77) have proposed algebraic approaches. Abstract data types are defined in a representation-independent specification in terms of the domains and ranges of operations. The meanings of operations are captured in axioms by stating their relationships to one another in a set of relations.

Standish (ST78) has taken a factored axiomatic approach to data structures wherein a set of "ground axioms"

addressing the pervasive underlying characteristics of data structures are formalized. Higher order axiom sets for structures such as trees and queues are then developed in such a manner that representations synthesized for the structures obey both the ground axioms and the higher order (structure) axioms. Given a well-understood set of ground axioms, one transforms the higher-order axioms to produce function definitions for the (higher order) structure and a data model for the ground axioms. This system thus provides a framework which has advantages for proving the correctness of programs and data representations. While this method provides a framework for a "factored" approach to correctness proofs, Guttag reports that it is not always easy to determine if the axiomatization is consistent and sufficiently complete. As the axioms must capture the semantics of the operations on the structures, the completeness criterion is especially important in the proof process.

Guha and Yeh (GY75) formalize the semantics of list structures using graph representations, and then define structure operations in terms of partial functions on graph configurations. This approach provides mathematical tools for the analysis of applications of list structures, but correctness of operations and implementation strategies are not addressed.

Of importance to the work presented in this paper is

the approach to data structures taken by Fleck (FA71), where list structures are formally shown to be identical to context free grammars. Fleck defines a list structure as a finite collection of sets of lists. The set of productions written for a particular recursive list then generates all possible instances of representations of the list structure. A construction is given by Fleck which shows the equivalence of context-free languages and list structure representations.

While the work of Fleck is significant, the work of Thompson (TJ81) actually forms the basis for the formalization of linked data structure operations addressed in the remainder of this paper. This approach combines graph theoretic and formal language concepts in developing grammatical descriptions of data structures. The next section introduces the approach to formal data structure specification. Additional discussion of the underlying theory is contained in Chapter 2.

1.4 Introduction to Formal Data Structure Specification.

Directed graphs are often used to communicate information pertaining to linked data structures. Because of the relative ease of expression, it is convenient to use K-formulas (BA75) to represent directed graphs. Briefly, graph nodes are represented in K-formulas by symbols such as single letters, and a prefix operator denotes an arc from one node to another. The relative positions of the operator

and node symbols specify the connectivity of the graph. A set of rules specifies all well-formed K-formulas.

K-formulas can be generalized to incorporate various types of links; for example, separate operator symbols can be used to distinguish the left and right links in a binary tree. The set of rules which describes well-formed K-formulas can be tailored for each particular data structure of interest, and expressed in the form of a context-free grammar (TJ81). Such a grammar generates strings of node and link symbols, such that strings representing all allowed data structures can be generated by the grammar, and only those strings.

To specify changes to data structures, one can describe the changes by showing how the K-formula changes. Concentrating on only those parts of the structure which are modified, the corresponding "before" and "after" substrings of the K-formula precisely describe the structural changes. This method of K-formula "transforms" is the approach to data structure operations taken in this paper.

In Chapter Two, the theory underlying the data structure operations is described. Formalized data structure operations are presented in Chapter Three, followed by examples of the application of the method in Chapter Four. Some thoughts on future work are given in Chapter Five.

CHAPTER TWO

2. Underlying Theory.

2.1 Introduction.

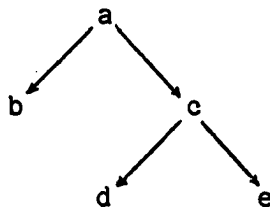
The formal specification of operations on linked data structures, as presented herein, is based on concepts from graph and language theory. Graphs are used as an intermediate descriptive tool; the K-formula method of representing graphs is fundamental to the entire balance of the paper. Grammars are used to define the allowable morphology of data structures, and properties of the particular type of grammars used guide the development of the method used to specify structure operations.

This chapter first addresses the use of K-formulas to represent directed graphs, and then applies grammatical methods to the generation of K-formulas. The resulting "K-grammars" are augmented to include node identification, and the correspondence between data structures and K-formulas is shown. Concepts from formal languages and automata are introduced, and applied in defining those data structures which are recognizable using the present methods. The material of Section 2.2 through Section 2.6.1 is a review of the work of others; the original contributions of this paper begin at Section 2.6.2.

2.2 Multirelational Graphs and K-formulas.

When discussing directed graphs, there is normally a single type of relation between nodes: node "a" is related to node "b" if there is an edge from "a" to "b", and nodes "a" and "b" are unrelated if there is no such directed edge. In data structures, it is frequently useful to distinguish links such as the "leftlinks" and "rightlinks" in a binary tree. For this reason, multirelational digraphs are introduced here.

DEFINITION: A multirelational digraph (see TJ81) $D = \langle A, R \rangle$ is an ordered pair of sets with A a set of sets of nodes and R a set of relations among the elements of the sets in A. As an example, the multirelational digraph $D = \langle \{\{a,b,c,d,e\}\}, \{\text{LEFT}, \text{RIGHT}\} \rangle$ with $\text{LEFT} = \{\langle a,b \rangle, \langle c,d \rangle\}$ and $\text{RIGHT} = \{\langle a,c \rangle, \langle c,e \rangle\}$ defines the binary tree shown below.



A convenient method of portraying multirelational digraphs involves the use of K-formulas.

A K-formula is used to represent the topology of a directed graph through the use of a sequence of symbols.

Two types of symbols are used: node symbols and operator symbols. An operator symbol preceding two node symbols denotes a directed edge from the first node to the second node. Thus, the K-formula $*ab$ describes a graph with nodes "a" and "b", and an edge from "a" to "b" denoted by the "*" operator.

In this paper, nodes are designated by (sometimes subscripted) letters a,b,c, . . . ; the lower case Greek letters (e.g., ρ, λ) are used as operator symbols. If more than one edge originates at a node, the corresponding K-formula contains as many K-operators preceding the node as there are edges originating at the node. Unique types of edges are denoted by unique edge symbols. For example, in a binary tree, " λ " may be used to represent the edge from a node to its left subtree (i.e., the left-link), and " ρ " to represent the edge from the node to its right subtree (i.e., the right link).

K-formulas may be combined using the "substitution rule" if common nodes are involved. If the K-formula A is defined to be ρab and the K-formula B is defined to be ρbc , we replace "b" of A with B to obtain $\rho a\rho bc$. Graphically,

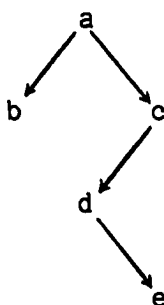
A: $a \rightarrow b$ and B: $b \rightarrow c$ combine to $a \rightarrow b \rightarrow c$.

Thus, the singly-linked list:

$a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f$

is represented by the K-formula $\rho a \rho b \rho c \rho d \rho e f$.

Similarly, the binary tree



is represented by the K-formula $\rho \lambda a b \lambda c \rho d e$.

The following recursive definition of K-formulas follows Berztiss (BA75).

- (a) A node symbol is a K-formula.
- (b) If A and B are K-formulas and λ is an operator, then λAB is a K-formula.
- (c) K-formulas are only those entities created under (a) and (b).

2.3 Grammars and Graph Grammars.

Since K-formulas are strings of symbols in which the symbols appear in a particular sequence, a tool is needed to construct well-formed sequences of symbols (sentences in a language) which meet the definition of K-formulas. A

grammar is a mathematical tool for generating languages. A grammar for a language L uses two disjoint sets of symbols: the set N of nonterminal symbols and the set T of terminal symbols. The set of terminal symbols is the alphabet over which the language is defined. The nonterminal symbols serve as placeholders in the generation of sentences of the language. The set P of formation rules, or "productions", describes how the sentences of the language are to be generated. A distinguished symbol S in N , called the "start symbol", is used to designate the productions which initiate the generation of sentences in the language defined by the grammar.

More formally, a grammar is defined as follows.

DEFINITION. A grammar is a 4-tuple $G = (N, T, P, S)$ where:

- (1) N is a finite set of nonterminal symbols, or variables.
- (2) T is a finite set of terminal symbols disjoint from N .
- (3) P is a finite set consisting of expressions (l, r) written in the form $l \rightarrow r$, where l is a string in $(N \cup T)^* N (N \cup T)^*$ and r is a string in $(N \cup T)^*$.
- (4) S is a symbol in N .

As an example, consider the following grammar.

$$G_1 = (\{A, B, S\}, \{a, b, c\}, P, S)$$

where P contains:

$$S \rightarrow \rho aA \quad (1)$$

$$A \rightarrow \rho bB \quad (2)$$

$$B \rightarrow c \quad (3)$$

The grammar G_1 generates the string " $\rho a \rho b c$ ", which is the K-formula for the singly-linked list $a \rightarrow b \rightarrow c$.

Before extensive discussion of grammars used to generate K-formulas ("K-grammars"), an introduction to graph-generating grammars ("graph grammars"; see TJ81) is presented. Because of the correspondence between graph grammars and K-grammars, one can work with either the pictures or text strings when studying directed graphs. Visualizing a linked data structure is easy when the corresponding graph is drawn.

Consider the grammar:

$$G_2 = (\{ \boxed{\text{s-list}} \}, \{ \bigcirc \rightarrow, \perp \}, P, \boxed{\text{s-list}}),$$

where P contains:

$$\boxed{\text{s-list}} \rightarrow \bigcirc \rightarrow \boxed{\text{s-list}}, \text{ and} \quad (1)$$

$$\boxed{\text{s-list}} \rightarrow \perp. \quad (2)$$

G_2 generates single-successor lists such as:

$$\perp$$

the null list generated by production (2);

$$\bigcirc \rightarrow \perp$$

a list with one node, generated by the production sequence (1),(2); and

$$\bigcirc \rightarrow \bigcirc \rightarrow \bigcirc \rightarrow \perp$$

a list with three nodes, generated by the production sequence (1)(1)(1)(2).

The productions in P are "redraw rules", which give all valid replacements of nonterminal symbols (such as $\boxed{\text{s-list}}$ in G_2).

Next consider the grammar:

$$G_3 = (\{ \boxed{B} \}, \{ \bigcirc, \bigcirc, \bigcirc, \bigcirc \}, P, \boxed{B}),$$

where P contains:

$$\boxed{B} \quad \text{-->} \quad \bigcirc \quad (1)$$

$$\boxed{B} \quad \text{-->} \quad \begin{array}{c} \bigcirc \\ | \\ \boxed{B} \end{array} \quad (2)$$

$$\boxed{B} \quad \text{-->} \quad \begin{array}{c} \bigcirc \\ | \\ \boxed{B} \end{array} \quad (3)$$

$$\boxed{B} \quad \text{-->} \quad \begin{array}{c} \bigcirc \\ / \quad \backslash \\ \boxed{B} \quad \boxed{B} \end{array} \quad (4)$$

G_3 generates binary tree graphs.

A graph grammar formally specifies all allowed graphs of a particular type. The nonterminal symbols in productions are placeholders for subgraphs. The productions, which are frequently recursive in that the same nonterminal appears on the left- and right-hand sides of the production, specify the generation rules. The terminal symbols appear in the resultant graph.

2.4 Grammars Augmented for Node Identification.

In directed graphs, the nodes are commonly assigned

identifying labels. Similarly, nodes of data structures can also be distinguished by node identifiers. In grammar G_1 above, the node identifiers are "a", "b", and "c". G_1 is capable of generating the sentence " $\rho a \rho b c$ ", and only that sentence, because specific node identifiers are incorporated in the productions of G_1 . What is desired is the capability to generate all instances of a particular type of data structure using a single grammar. To achieve this capability, the grammar must be able to generate the K-formulas which correspond with all instances of the data structure involving the nodes specified by the terminal symbols in the grammar.

To generate the required sequences of terminal symbols, Thompson (TJ81) has augmented grammars as described in the following **DEFINITION**. Let $A \rightarrow \langle\langle\beta\rangle\rangle$ be a production in a grammar. The string β enclosed by French quotes " $\langle\langle$ " and " $\rangle\rangle$ " is called a phrase. Let the string β contain a phrase indeterminate symbol of the form:

$$a_{[i]}.$$

The set I of node phrase indeterminates is mapped into the set T of terminal symbols by the definitive mapping $M: I \times Q \rightarrow T$, where Q is a set of integers which identify distinct phrases.

The grammar

$$G_4 = (\{A\}, \{\rho, a, b, c, d\}, P, A)$$

augmented by the mapping

$$M4 = ([(1,1), a], [(2,2), b], [(3,3), c], [(4,4), d]),$$

with P given by

$$A \rightarrow \langle \langle \rho a_{[i]} A \rangle \rangle_q \quad (1)$$
$$A \rightarrow \langle \langle a_{[i]_q} \rangle \rangle \quad (2)$$

generates the singly-linked lists shown by the following production sequence.

```

A --> <<pa      A>>
      [i]      1

--> <<pa      <<pa      A>> >>
      [i]      [i]      2      1

--> <<pa      <<pa      <<pa      A>> >> >>
      [i]      [i]      [i]      3      2      1

--> <<pa      <<pa      <<pa      <<a      >> >> >> >>
      [i]      [i]      [i]      [i]      4      3      2      1

```

Applying M4, where "i" is the nesting depth, removes the French quotes as follows:

```

<<pa    <<pa    <<pa    d>> >> >>
      [i]      [i]      [i]      3  2  1

<<pa    <<pa    pcd>> >>
      [i]      [i]      2  1

<<pa    pbbcd>>
      [i]      1

pappbcd.

```

Applying another mapping M4.1 when using the same grammar results in another structure of the same type. For example, suppose

$$M4.1 = ([(1,1), d], [(2,2), c]).$$

Then the production sequence

$$\begin{aligned} A &\rightarrow \langle \rho a \quad A \rangle \\ &\quad [i] \quad 1 \\ &\rightarrow \langle \rho a \quad \langle a \quad \rangle \rangle \\ &\quad [i] \quad [i] \quad 2 \quad 1 \end{aligned}$$

generates the singly-linked list ρdc .

To expand the utility of Thompson's augmented grammars, the phrase indeterminates are extended here to include link symbols. Productions may then incorporate both link and node "metasymbols" inside the French quotes. The metasymbols are resolved by a modified mapping such as:

$$J \times K \times Q \rightarrow L \times N$$

where J is the set of link phrase indeterminate symbols, K is the set of node phrase indeterminate symbols, L is the set of link symbols and N is the set of node symbols. This extension is useful in defining the class of linked data structures to be transformed.

2.5 Context-Free Languages and Recognizers.

Using K -formulas to represent linked data structures offers the important qualities of notational conciseness and precision. A K -grammar provides a mechanism for generating the K -formulas which correspond with all allowed data structures defined by the grammar. Care in the choice of a data structure K -grammar allows one to take advantage of standard recognition tools in addition to the grammatical generation tools. In particular, if the K -grammar is context-free, then a push-down automaton (PDA) can be used

to recognize whether a particular K-formula is well-formed according to the rules of the K-grammar.

The following definition is based on Aho and Ullman (AU72). **DEFINITION.** Let $G = (N, T, P, S)$ be a grammar. G is a context-free grammar if each production in P is of the form $A \rightarrow \beta$, where A is a nonterminal in N and β is a (possibly empty) string of nonterminals and terminals in $(N \cup T)^*$. The term "context-free" is appropriate because in a derivation of a particular sentence in the language $L(G)$ generated by G , a nonterminal "A" can be replaced by the right-hand-side of a production (whose left-hand-side is "A") without regard for the symbols which precede and follow "A". Grammar G_1 in paragraph 2.3 above is a context-free grammar. Grammar G_4 in paragraph 2.4 above is not context-free because the right-hand-sides of productions contain symbols not in $(N \cup T)^*$; although the "supplementary" symbols are eventually removed, additional "machinery" in the form of a mapping function is needed to do so.

Given an instance of a K-formula, one can determine if it is well-formed according to the rules of a particular context-free K-grammar by use of a recognizer termed a pushdown automaton (PDA). A PDA consists of a read-only input medium, a finite state control and an auxiliary memory called a pushdown list. The recognizer operates by making a sequence of moves, where a move involves reading an input

symbol, and based on the input symbol, the state of the finite control and the contents of the pushdown list (stack):

(1) shifting the input head right to the next symbol on the input medium, or keeping the input head stationary;

(2) revising the contents of the stack; and

(3) changing the state of the finite control.

The activities of a PDA recognizer can be described by "configurations" of the recognizer, which include:

(1) the state of the finite control;

(2) the content of the stack; and

(3) the location of the input head, and the unused contents of the input medium.

The initial configuration is one where the finite control is in a specified initial state, the input head is at the leftmost symbol and the memory has specified initial contents. The final configuration of the recognizer is one where the finite control is in a final state, the input has been exhausted and the stack is empty.

More formally, a PDA R is defined (AU72) as a 7-tuple as follows. **DEFINITION:** $R = (Q, A, G, d, q_0, Z_0, F)$, where:

(1) Q is a finite set of state symbols, giving the possible states of the finite state control,

(2) A is a finite input alphabet,

(3) G is a finite alphabet of stack symbols,

(4) d is a mapping of $Q \times (A \cup \{e\}) \times G$ to the finite subsets of $Q \times G^*$ ("e" represents the empty string),

(5) q_0 in Q is the initial state of the finite control,

(6) Z_0 in G is the symbol that appears initially on the stack, and

(7) F contained in Q is the set of final states.

A configuration of P is a triple (q, w, g) in $Q \times A^* \times G^*$, where:

(1) q represents the current state of the finite control,

(2) w represents the unused portion of the input, and

(3) g represents the contents of the stack. The leftmost symbol of g is the topmost stack symbol.

A "move" by R is represented by the binary relation $|--$, for example:

$$(q_1, aw, b\beta) |-- (q_2, w, ab\beta)$$

if $d(q_1, a, b)$ contains (q_2, ab) .

A string " w " is accepted by R if $(q_0, w, Z_0) |^*- (q, e, g)$ for some q in F and g in G^* , or if $(q_0, w, Z_0) |^*- (q, e, e)$ for some q in Q . The language $L(R)$ defined by R is the set of strings accepted by P .

Aho and Ullman (AH72) have shown that the languages recognized by PDAs are exactly the context-free languages. This means that, given a context-free grammar $G = (N, T, P, S)$, we can construct a PDA R such that $L(R) = L(G)$; the construction is:

$$R = (\{q\}, T, \{N \cup T\}, d, q, S, \emptyset)$$

where "d" is defined as follows:

(1) If $A \rightarrow \beta$ is in P , then $d(q, e, A)$ contains (q, β) .

(2) $d(q, a, a) = \{(q, e)\}$ for all a in T .

As an example, consider the grammar G_1 of Section 2.3 above. The following PDA recognizes sentences of G_1 .

$$R = (\{q\}, \{a,b,c,\rho\}, \{a,b,c,\rho,S,A,B\}, d, q, S, 0)$$

where:

$$d(q, e, S) = (q, \rho aA)$$

$$d(q, e, A) = (q, \rho bB)$$

$$d(q, e, B) = (q, c)$$

$$d(q, a, a) = (q, e)$$

$$d(q, b, b) = (q, e)$$

$$d(q, c, c) = (q, e)$$

$$d(q, \rho, \rho) = (q, e).$$

The following configuration sequence shows that $(q, \rho a \rho bc, S) \vdash^* (q, e, e)$.

$$\begin{aligned} (q, \rho a \rho bc, S) &\vdash (q, \rho a \rho bc, \rho aA) \\ &\vdash (q, a \rho bc, aA) \\ &\vdash (q, \rho bc, A) \\ &\vdash (q, \rho bc, \rho bB) \\ &\vdash (q, bc, bB) \\ &\vdash (q, c, B) \\ &\vdash (q, c, c) \\ &\vdash (q, e, e) \end{aligned}$$

2.6 Recognizable Structures.

A pushdown automaton (PDA) can be used to recognize a linked data structure if the structure has certain characteristics. In this section, some concepts from graph theory are briefly examined, and the properties which render a data structure recognizable are enumerated.

2.6.1 Eulerian Graphs. A traversal of a data structure is defined as a systematic search in which each node of the structure is visited. A traversal of a syntactically-specified data structure is represented by a K-formula derivable from the data structure grammar. A K-formula identifies not only the nodes and connectivity of an instance of a data structure, but also a traversal of the structure. A desirable property is to be able to visit all nodes by following a traversal path which traces each edge exactly once. The following definitions follow Chen (CW71). A graph is connected if every pair of its nodes are connected. The degree $d(i)$ of a node "i" is equal to the number of edges incident with "i". An edge train is a sequence of edges in which all edges are distinct. A closed edge train containing all the edges of a directed graph is called an Euler line of the directed graph. A connected graph is an Euler line if, and only if, the degree of each of its nodes is even. Another view of the Euler line is that a connected graph is a directed Euler line if, and only if, the number of edges entering a node "i" (the indegree of

"i") is equal to the number of edges leaving "i" (the outdegree of "i") for all nodes "i".

2.6.2 K-Formulas for Eulerian Graphs. The K-formulas for graphs which are Eulerian have certain properties. Because an Euler line consists of a closed edge train, the associated K-formula must begin and end with the same node symbol; the symbol "h" is normally used to identify this distinguished header node. That all edges are distinct in an Euler line requires that the symbols

$$\begin{array}{ccc} l & a & a \\ & i & j \quad k \end{array}$$

not be replicated in the K-formula for any given triple (i,j,k). ("l" is a link symbol; "a" is a node symbol.) To preclude multiple definition of any particular link, the symbols

$$\begin{array}{cc} l & a \\ & i \quad j \end{array}$$

may not be replicated in the K-formula for any given pair (i,j). With application of the substitution rule for K-formula combination, the aggregate result of these constraints on K-formulas which represent Euler lines is that they must conform to the following expression.

$$\begin{array}{ccc} l & h \ll l & a >> *h \\ & i & [j] [k] \quad q \end{array} \quad (2.6.3.1)$$

The "*" operator indicates closure, which means that the quantity inside the French quotes is repeated zero or more times. The French quotes prohibit recurrence of any given

link-node pair. (Note that the definitive mapping:

$$J \times K \times Q \rightarrow L \times N$$

may, and usually does, allow replication of a link symbol. While a node label may recur, it may appear only as many times as there are edges incident to the corresponding node, and its appearance must always conform to the relevant data structure grammar. Note that the mapping may not generate the link-node symbol pair "lh".) In the case of one or more data structures embedded within a data structure or structures, a node symbol may map to the header node(s) of the embedded structure(s).

2.6.3 Eulerian Graphs and Recognizable Structures. To determine whether an instance of a data structure conforms to a certain data structure K-grammar, one can apply automata theory. In this section we show that if a graph G is Eulerian, then there is a traversal of G whose K-formula is PDA recognizable. The proof involves three theorems which use the following generalized grammar.

$$E = (N, T, P, A)$$

where $N = \{A, B\}$

$T =$ sets of link and node symbols
denoted below by $l_{[i]}$, $l_{[j]}$,
 $a_{[k]}$, h , and $h_{[k]}$. The $h_{[k]}$
are substructure header nodes,
which are unique for each
substructure and distinct
from "h".

P is given by:

$$A \rightarrow \langle \langle 1 \quad hBh \rangle \rangle \\ [i] \quad q$$

$$A \rightarrow \langle \langle 1 \quad hh \rangle \rangle \\ [i] \quad q$$

$$B \rightarrow \langle \langle 1 \quad a \quad \rangle \rangle \\ [i] [k] \quad q$$

$$B \rightarrow \langle \langle 1 \quad a \quad B \rangle \rangle \\ [i] [k] \quad q$$

$$B \rightarrow \langle \langle 1 \quad h \quad B1 \quad h \quad \rangle \rangle \\ [i] [k] [j] [k] \quad q$$

With suitable mapping functions, E generates the K-formulas which correspond with data structures with Eulerian traversals. Hereinafter, such data structures are termed "type-E" data structures.

In E, note that the set of actual terminal symbols is determined by application of mapping functions which specify the link and node symbols. The mapping functions must assure that a link symbol at any given node is not multiply-defined. The mappings may allow node symbols to be replicated in a manner which conforms to the data structure grammars. The grammar E is important in the following theorem.

Theorem 1. For a graph G, there exists a traversal of G whose K-formula is derivable from the grammar E if and only if G is Eulerian.

Proof: (If): Because G is Eulerian, there exists a closed edge train which begins and ends with the distinguished node "h". Adjacent edges in this edge train are related such

that if an edge from "a" to "b" exists, then there is another edge from "b" to some "c" for all nodes in the graph. As all edges in Eulerian G are distinct, one can apply the K-formula substitution rule to generate a K-formula which consists of zero or more link-node pairs, all of which is prefixed by a link symbol followed by the header node label, and all of which is suffixed by the header node label. This is exactly the K-formula pattern generated by the grammar E.

(Only if): Suppose G is not Eulerian. Then there exist at least two nodes whose degree is odd, because the total degree of G (the sum of the indegree and outdegree of all nodes) must be an even number; this is true because each edge in G contributes by 2 to the total degree. If the degree of a node "y" is odd, then one of the following holds.

- (a) The indegree of "y" is one and outdegree is zero. This condition results in a K-formula "lxy", and there exists no K-formula "lyz" because no edges leave "y". Therefore, the substitution rule cannot be applied to "lxy", and the resultant K-formula does not conform to the grammar E.
- (b) The indegree of "y" is zero and the outdegree of "y" is greater than zero. In this case, there are no edge representations of the form

"lxy", but there is one or more of the form "lyz". The "lyz" cannot be combined with an "lxy", and the resultant K-formula cannot be derived from E.

- (c) The indegree of "y" is greater than the outdegree of "y" which is greater than zero. This condition results in at least one more K-formula of the form "lxy" than there are K-formulas of the form "lyz". Thus, at least one of the "lxy" cannot be combined with "lyz" and the resultant K-formula cannot be derived from the grammar E.
- (d) The outdegree of "y" is greater than the indegree of "y" which is greater than zero. In this case, there exists at least one more K-formula of the form "lyz" than there are K-formulas of the form "lxy". Therefore, at least one of the "lyz" cannot be combined with "lxy" and the resultant K-formula cannot be derived from the grammar E.

Given a traversal of an Eulerian graph G, which traversal is described by a K-formula derivable from E, we next show that the K-formula is PDA-recognizable.

Theorem 2: Any K-formula derived from the grammar E is PDA-recognizable.

Proof: The proof is given by constructing a PDA which recognizes K-formulas which are derived from E. Such K-formulas begin with a link symbol followed by the header node symbol, followed by zero or more link-node pair symbols, followed finally by the header node symbol.

Define: $A'(\&)$ to be a PDA which initially has an empty stack.

$B'(z)$ to be a PDA which has "z" at the top of its stack when invoked.

$q:B'(z)$ to indicate that the PDA which invoked B' makes a transition to state q if $B'(z)$ terminates.

The PDA of interest consists of two components, $A'(\&)$ and $B'(z)$, as defined above. A' recognizes the structure header node, and invokes B' to recognize interior nodes and substructures. Three distinct link symbols are assumed here; additional link symbols merely result in additions to the mappings d^\sim and $d^{\sim\sim}$. Using the PDA definition of Section 2.5, the following apply.

$$A' = (Q^\sim, A^\sim, G^\sim, d^\sim, q_{10}, \&, q_f)$$

where:

$$Q^\sim = \{q_{10}, q_{11}, q_{12}, q_{13}, q_f\}$$

$$A^\sim = \{l_i\} \cup \{a_k\} \cup \{h_k\} \cup \{h\}$$

$$G^\sim = \{\&, h\}$$

d^\sim is the mapping defined as follows.

$$d^{\sim}(q_{10}, l_1, \&) = (q_{11}, \&) \quad A'1$$

$$d^{\sim}(q_{10}, l_2, \&) = (q_{11}, \&) \quad A'2$$

$$d^{\sim}(q_{10}, l_3, \&) = (q_{11}, \&) \quad A'3$$

$$d^{\sim}(q_{11}, h, \&) = (q_f : B'(h), \&) \quad A'4$$

$$d^{\sim}(q_{11}, h, \&) = (q_{12}, h) \quad A'5$$

$$d^{\sim}(q_{12}, h, h) = (q_f, \&) \quad A'6$$

$$B' = (Q^{\sim\sim}, A^{\sim}, G^{\sim\sim}, d^{\sim\sim}, q_0, h, q_t)$$

where:

$$Q^{\sim\sim} = \{q_0, q_1, q_2, q_3, q_t\}$$

A^{\sim} is as in A' above.

$$G^{\sim\sim} = \{h\} \cup \{h_k\}$$

$d^{\sim\sim}$ is the following mapping.

$$d^{\sim\sim}(q_0, l_1, h) = (q_1, h) \quad B'1$$

$$d^{\sim\sim}(q_0, l_2, h) = (q_1, h) \quad B'2$$

$$d^{\sim\sim}(q_0, l_3, h) = (q_1, h) \quad B'3$$

$$d^{\sim\sim}(q_1, a_k, h) = (q_2, h) \quad B'4$$

$$\begin{aligned}
d^{--}(q_1, h_k, h) &= (q_2, h_k h) & B'5 \\
d^{--}(q_2, l_1, h) &= (q_3, h) & B'6 \\
d^{--}(q_2, l_2, h) &= (q_3, h) & B'7 \\
d^{--}(q_2, l_3, h) &= (q_3, h) & B'8 \\
d^{--}(q_2, l_1, h_k) &= (q_3, h_k) & B'9 \\
d^{--}(q_2, l_2, h_k) &= (q_3, h_k) & B'10 \\
d^{--}(q_2, l_3, h_k) &= (q_3, h_k) & B'11 \\
d^{--}(q_3, a_k, h) &= (q_2, h) & B'12 \\
d^{--}(q_3, a_k, h_k) &= (q_2, h_k) & B'13 \\
d^{--}(q_3, h_k, h) &= (q_2, h_k h) & B'14 \\
d^{--}(q_3, h_i, h_i) &= (q_2, e) & B'15 \\
d^{--}(q_3, h_i, h_j) &= (q_2, h_i h_j) & B'16 \\
d^{--}(q_2, h, h) &= (q_t, e) & B'17
\end{aligned}$$

In the above, the mapping notation $d^{--}(q, x, y) = (q, e)$ denotes that the top of the stack is removed; the mapping notation $d^{--}(q, x, y) = (q', xy)$ denotes that the symbol "x" is stacked on top of the previous top-of-stack symbol "y".

By Theorem 1, G is Eulerian, and this condition assures that the corresponding K-formula is finite in length. With a finite input, A' either halts in the final state, or halts

in an error state. If A' halts in its final state, the proof of the next theorem shows that its input conforms with E.

Theorem 3: If A'(&) terminates in its final state, then the K-formula which was its input is derivable from E.

Proof: The grammar E generates K-formulas which are described by the following expression:

$$l \ h \langle l \ a \ \rangle \ *h \\ i \quad [i] \ [k] \ q$$

where l is a valid link symbol.
[i]

The a node symbols may include
[k]
header nodes of substructures.

The French quotes and associated mapping assure that node symbols, if repeated, occur only in an allowed sequence. We must prove that input sequences derivable from the above expression, and only those, cause A'(&) to terminate. The method used here consists of showing that A'(&) terminates if the input is derivable from:

$$l \ h(l \ a \) \ *h \\ i \quad i \ k$$

and then showing what conditions must exist to ensure that the input is also derivable from:

$$l \ h \langle l \ a \ \rangle \ *h. \\ i \quad [i] \ [k] \ q$$

The proof proceeds by considering state sequences of the machines A' and B' of Theorem 2, and examining the input which produced those state sequences. In the following, the notation [A'1,A'2,A'3,...] denotes application of one of the

bracketed transitions of machine A'; similar symbol sequences apply to transition sequences for machine B'.

Case 1: A' follows the transition sequence [A'1,A'2,A'3], A'5, A'6. The input K-formula consisted of a link symbol followed by "hh", which is certainly derivable from E.

Case 2: A' follows the transition sequence [A'1,A'2,A'3], A'4 and invokes B'. The input already processed when B' is invoked is a link symbol followed by "h". If B' is to halt successfully and return A' to a final state, B' must follow the following state sequence:

$$\begin{array}{ccccccc} q & q & q & (q & q) & *q & \\ 0 & 1 & 2 & 3 & 2 & t & \end{array}$$

The following input must exist to produce the above state sequence:

- (a) A link symbol to transition from
 q_0 to q_1 using [B'1,B'2,B'3]; and
 followed by
- (b) a node symbol to transition from
 q_1 to q_2 using [B'4,B'5]; and
 followed by
- (c) zero or more instances of:
 - (1) a link symbol to transition
 from q_2 to q_3 using
 [B'6,B'7,B'8,B'9,B'10,B'11];
 and

(2) a node symbol to transition
 from q_3 to q_2 using
 $[B'_{12}, B'_{13}, B'_{14}, B'_{15}, B'_{16}]$; and
 followed by

(d) the header node symbol "h" to
 transition from q_2 to q_t .

The input of (a)(b)(c)(d) above to B' is represented by

$$l a (l a)^* h.$$

$$i k i k$$

When the inputs to A' and B' are combined, the aggregate
 result is

$$l h (l a) h,$$

$$i i k$$

which is derivable from E. The combined result of Case 1
 and Case 2 is that the input to A' and B' is of the form:

$$l h (l a)^* h$$

$$i i k q$$

That no other symbol sequence causes A' to reach its final
 state is apparent by inspection of the state transitions of
 A' and B' .

We have now shown that if $A'(\&)$ terminates, then the
 input consists of alternating link and node symbols,
 followed by the header node symbol. The first (leftmost)
 node symbol is that of the header node. We now show that a
 given link-node symbol pair can appear only once in the
 input K-formula.

If a certain link-node symbol pair "lx" is replicated

in a K-formula, then either the node "x" is repeated within a substructure (subcase 2a), or the node "x" is common to two substructures (subcase 2b).

Subcase 2a: Suppose a node "x" is repeated within a substructure. Then the input K-formula contains a string of the form:

. . . lxy . . . lxz . . .

Because no link may be multiply-defined, "y" and "z" are one and the same node. Thus, the link from "x" to "y" is retraced in the traversal which the K-formula describes, and there is a loop in the corresponding structure; this loop may be retraced an infinite number of times. But the looping of the traversal implies that A'(&) does not halt, contradicting the theorem statement. Therefore, no node "x" may be repeated within a substructure.

Subcase 2b: Assume two substructures contain a common node. (Recall that the header node of each substructure must be unique, and thus the common node is not the header node.) To share a node, the node formats of both substructures must be identical, including link definitions. Suppose that B' successfully recognized the substring of the K-formula (a "K-string") which corresponds with the substructure whose header node is "h1". Then, while processing the K-string which represents the other substructure in which the header node is "h2", B' will have "h2" at the top of its stack when it encounters "h1" in the input; "h2" will not be visited

again in the traversal, which means B' cannot terminate with an empty stack. Thus, A' cannot terminate and we have a contradiction of the theorem statement. We have now shown that a link-node symbol pair cannot be repeated in the input K-formula, and thus the input K-formula is derivable from E .

Theorem 3 assures that if A' terminates, then its input K-formula is derivable from E and the corresponding data structure is Eulerian. A proof shorter than that given above is based on expression (2.6.3.1). A' terminates only if the input is of the form:

$$\begin{array}{c} l \ h(l \ a \)^*h. \\ i \ \quad j \ k \end{array}$$

A K-string such as:

$$\begin{array}{c} l \ h l \ a \ l \ a \ h \\ 1 \ \quad 1 \ 1 \ 2 \ 2 \end{array}$$

can be expanded to a series of K-formulas:

$$\begin{array}{c} l \ h a \ ; \ l \ a \ a \ ; \ l \ a \ h. \\ 1 \ \quad 1 \ \quad 1 \ 1 \ 2 \ \quad 2 \ 2 \end{array}$$

By inspection, the indegree equals the outdegree at each node in the corresponding data structure. Because a graph is Eulerian if and only if the indegree equals the outdegree at each node in the graph, the data structure is Eulerian and the corresponding K-formula is derivable from E .

Theorems 1, 2 and 3 collectively show that K-formulas which correspond with Eulerian graphs are PDA-recognizable. The recognizable K-formulas are those which can be derived from the grammar E ; the corresponding structures are termed "Type-E" structures. Given a grammar for a data structure,

to answer the PDA-recognizability question one must determine if the data structure K-grammar generates K-formulas which correspond with the grammar E. That is, one must show that the K-grammar generates K-formulas which can be described by the expression (2.6.3.1), repeated here for convenience.

$$1 \ h \langle \langle 1 \quad a \quad \rangle \rangle \ *h \\ i \quad [i] \ [i] \ q$$

Consider the grammar:

$$G5 = (\{A, B\}, \{\rho, h, a_1, a_2, \dots\}, P, A)$$

where P is given by:

$$A \rightarrow \rho h B \quad (1)$$

$$B \rightarrow \langle \langle \rho a \quad B \rangle \rangle \\ [i] \quad q \quad (2)$$

$$B \rightarrow h \quad (3)$$

and the mapping function:

$$I \times Q \rightarrow T$$

which maps $\langle \langle a \quad \rangle \rangle$ to a .
 $[i] \quad q \quad q$

To show that grammar G5 generates K-formulas which conform to the grammar E, and thus that the corresponding data structures are Eulerian, first note that all node symbols (except "h") in K-formulas generated by G5 are unique. Thus, there can be no link-node symbol pair replication. Next, note that the production sequences ("pi-strings") of G5 are described by (1)(2)*(3). The mandatory use of productions (1) and (3) ensures that the

resulting K-formulas begin with a link symbol followed by the header node symbol, and end with the header node symbol. Application of production (2) always generates a unique link-node symbol pair. Therefore, K-formulas generated by G5 are PDA-recognizable.

More formally, the pi-strings of G5 generate K-formulas which are described by the following expression:

$$\rho h \langle \langle \rho a \quad \rangle \rangle * h. \\ [k] \quad q$$

This expression conforms to (2.6.3.1) above.

2.6.4 Observations on Recognizability. The grammar E of Section 2.6.3 is not a context-free grammar. Indeed, no grammar which contains phrase indeterminates is context-free because resolution of a phrase indeterminate symbol depends upon the context (e.g., nesting level) in which the symbol appears. The phrase indeterminates and associated mappings are necessary to ensure that link-node symbol pairs are not replicated, and in conjunction with the grammar, serve to specify the traversal of the structure.

A PDA can be used to recognize well-formed K-formulas, and is constructed from the data structure grammar. The PDA must be augmented, however, to ensure that all incorrect usages of node symbols are detected. In particular, augmentation is required to detect illegal recurrence of a node symbol. The PDA may also fail to halt in certain cases, such as the event of a loop in a traversal, which is

reflected as a traversal of infinite length. Modification of a PDA can incorporate a mechanism to detect potential traversal loops.

The data structure grammars are not strictly context-free, and the automata used to recognize K-formulas are not strictly PDAs. The formalisms are, however, very close to what is required, and with minor modifications, provide very useful machinery for automating data structure modifications.

We have now established the theoretical basis for formally describing operations on Eulerian data structures. In the next chapter, data structure transformations are examined, and a transform syntax is given.

CHAPTER THREE

3. Formalized Data Structure Operations.

3.1 K-Grammars for Several Common Data Structures.

In this chapter, the theory for formalized operations on data structures is developed. We begin by examining the K-grammars for some common structures. The grammar G4 of Section 2.4, in concert with a mapping function, generates K-formulas for a singly-linked list. Consider next the grammar

$$\text{SLL} = (\{A, B\}, \{\rho, h, a, b, c, d, e, i, j, k, \dots, z\}, P, A)$$

where P is given by:

$$A \rightarrow \rho h B \quad (1)$$

$$B \rightarrow \langle \langle \rho a \quad B \rangle \rangle_{[i] \quad q} \quad (2)$$

$$B \rightarrow h \quad (3)$$

SLL, augmented by a mapping function, generates K-formulas for circular singly-linked lists. The PDA-recognizability of the SLL K-formulas has been established in the discussion of G5 in the previous chapter.

K-formulas for doubly-linked circular lists can be generated by the grammar

$$\text{DLL} = (\{A, B\}, \{\rho, \lambda, a, b, c, d, e, i, j, \dots, z\}, P, A)$$

where P is given by:

$$A \rightarrow \rho h B h \quad (1)$$

$$B \rightarrow \langle \langle \rho a \quad B \lambda a \quad \rangle \rangle \quad (2)$$

$\begin{matrix} [i] & [i] & q \end{matrix}$

$$B \rightarrow \lambda h \quad (3)$$

Once again, a mapping function is required to resolve the phrase indeterminates.

In the DLL productions, " ρ " denotes the "right" or "forward" link of the doubly-linked list, and " λ " denotes the "left" or "backward" link. The same phrase indeterminate symbol appears twice in the second production--once after the forward link and once after the backward link. A DLL K-formula such as:

$$\rho h \rho a \rho b \lambda h \lambda b \lambda a h$$

not only describes a list with nodes "h", "a" and "b", but also specifies a traversal of the list. In this case, one follows the forward link " ρ " from "h" to "a", the " ρ " link from "a" to "b", the " ρ " link from "b" to "h", and then follows the backward link " λ " from "h" to "b", the " λ " link from "b" to "a", and finally the " λ " link from "a" to "h". Graphically, the above traversal is:

$$h \rightarrow a \rightarrow b \rightarrow h \rightarrow b \rightarrow a \rightarrow h.$$

To show that the K-formulas generated by DLL are PDA-recognizable, notice that the allowable production sequence is (1)(2)*(3). The resulting K-formula is described by:

$$\begin{array}{cccccccccccc} \rho h \rho a & \rho a & \dots & \rho a & \rho a & \lambda h \lambda a & \lambda a & \dots & \lambda a & \lambda a & h. \\ 1 & 2 & & n-1 & n & n & n-1 & & 2 & 1 \end{array}$$

Production (2) and the phrase indeterminate mapping ensure that link-node symbol pairs are not replicated, if the header node label "h" is disallowed in the mapping. Therefore, DLL generates K-formulas which are described by:

$$\rho h \langle \langle l \quad a \quad \rangle \rangle * h, \\ [i] [k] \quad q$$

which conforms to (2.6.3.2). These K-formulas are PDA-recognizable.

The following grammar RTBT generates K-formulas for right-threaded binary trees.

$$\text{RTBT} = (\{A, B\}, \{\rho, \lambda, \rho^{\sim}, a, b, c, d, e, i, j, \dots, z\}, P, A)$$

where P is given by:

$$A \rightarrow \lambda h B h \quad (1)$$

$$A \rightarrow \rho^{\sim} h h \quad (2)$$

$$B \rightarrow \langle \langle \lambda a \quad B \rho a \quad B \rangle \rangle \\ [i] [i] \quad q \quad (3)$$

$$B \rightarrow \langle \langle \lambda a \quad B \rho^{\sim} a \quad \rangle \rangle \\ [i] [i] \quad q \quad (4)$$

$$B \rightarrow \langle \langle \rho a \quad B \rangle \rangle \\ [i] \quad q \quad (5)$$

$$B \rightarrow \langle \langle \rho^{\sim} a \quad \rangle \rangle \\ [i] \quad q \quad (6)$$

A mapping function is used to resolve the phrase indeterminates.

In the RTBT productions, " λ " denotes the left link at a node, which is followed to reach the left subtree of the

node. The right link is denoted by " ρ ", which points to the right subtree of a node. If there is no right subtree at a particular node, then the right link at that node points to the successor of the node when an inorder traverse (HS76) is used; in the K-formula, this pointer, or "thread", is signified by the symbols " $\rho\sim$ ". Thus, the grammar RTBT above generates K-formulas which not only provide the topology of right-threaded binary trees, but also give the inorder traversal of these trees; the list of nodes which corresponds with an inorder traverse is obtained from the K-formula by printing all node labels which do not immediately follow a left link (" λ ") symbol.

The K-formulas generated by RTBT are PDA-recognizable, shown as follows. The allowable production sequences are (2) alone, which conforms with (2.6.3.1), and (1) followed by the appropriate selection of the productions (3) through (6) which finally results in resolution of all non-terminal symbols. Productions (3) and (4) disallow replication of link-node symbol pairs, and the phrase indeterminate mapping ensures that node labels generated by any single production are distinct from those generated by any other production. All of the terminal symbols generated by (3) through (6) are link-node symbol pairs. Therefore, RTBT generates K-formulas which begin with a link symbol followed by the header node symbol, followed by zero or more link-node symbol pairs, and terminated by the header node symbol.

This is exactly the pattern generated by (2.6.3.1), and thus RTBT generates PDA-recognizable K-formulas.

3.2 The Correspondence of K-Grammars and Classical Data Structure Generation.

3.2.1 Introduction.

K-grammars generate K-formulas which describe the topology and traversal of linked data structures. As introduced in Chapter One, the most prevalent method of providing this information in the past has involved the use of a programming language and supplementary diagrams. In this section, the correspondence between this classical method and the present approach is described.

A K-grammar is a generator of K-formulas, and can be compared with a programming language algorithm which constructs a particular type of data structure. Consider an algorithm which constructs right-threaded binary trees by adding nodes in a way such that a newly-inserted node would be the last node seen in a depth-first traversal (BA75). A C-language example of such an algorithm is shown in Figure 3.2.1. Discussion of the correspondence of this algorithm and the K-grammar method requires the following **DEFINITION:** A left-most derivation of a K-formula is one in which the left-most nonterminal symbol in a sentential form is replaced before other nonterminal symbols.

```

/* rtbt constructor algorithm */
#define printnod printf("%c", t.label[n]);
#define NE 15      /* number of elements in rtbt */
#define NULL 999   /* flag for null index */
#define HEAD 1     /* index of head node */
#define YES 1
#define NO 0
#define FOREVER 1
#define ERROR 99999 /* error symbol */
#define SUCCESS 0
int debug = NO;    /* flag to enable debug messages */
struct rtbt {
    int llink[NE];
    int rlink[NE];
    char label[NE];
    int mark[NE];
} t;
main()
{
    int retcode;      /* return code for consrtbt */
    t.llink[HEAD] = NULL;
    t.rlink[HEAD] = -1;
    t.label[HEAD] = 'h';
    consrtbt(2, 'a', 'h', 'L');
    consrtbt(3, 'b', 'a', 'L');
    consrtbt(4, 'c', 'a', 'R');
    consrtbt(5, 'd', 'c', 'R');
    consrtbt(6, 'e', 'd', 'L');
    consrtbt(7, 'f', 'd', 'R');
    consrtbt(8, 'g', 'f', 'L');
    consrtbt(9, 'i', 'g', 'L');
    consrtbt(10, 'j', 'i', 'L');
    consrtbt(11, 'k', 'i', 'R');
    consrtbt(12, 'm', 'k', 'R');
    printf ("%s", "n");
} /* end of main */
consrtbt (indexin, nodelabl, parent, linktype)
int indexin; /* index of node to be inserted */
char nodelabl; /* label of inserted node */
char parent; /* label of node which will be parent */
char linktype; /* which points to inserted node */
{
    int n; /* index of node presently visited */
    int c; /* index of last node visited */
    int i; /* loop control index */
    int l; /* last node reached via other */
    printf("n"); /* than thread */
    for (i = 0; i < NE; i = i + 1) t.mark[i] = NO;
}

```

Figure 3.2.1: Binary Tree Constructor Algorithm
(Page 1 of 3)

```

if (t.llink[HEAD] == NULL && parent == 'h')
    { if (debug) printf ("%s", "Qhhn"); /* A --> Qhh */
      c = HEAD;
    }
else
    { n = t.llink[HEAD]; /* A --> LhBh */
      if (debug) printf ("%s", "*** tree not empty n");
      l = NULL;
      while (n != HEAD)
          { c = n;
            if (t.llink[n] != NULL && t.rlink[n] > 0) /* B --> LaBRaB */
                { if (debug) printf ("%s", "*** LaSRaT n");
                  if (t.mark[n] != YES)
                      { t.mark[n] = YES;
                        printnod;
                        n = t.llink[n];
                      }
                  else
                      { printnod;
                        n = t.rlink[n];
                        l = n;
                      }
                }
            else if (t.llink[n] != NULL && t.rlink[n] < 0) /* B --> LaBQa */
                { if (debug) printf ("%s", "*** LaSQan");
                  if (t.mark[n] != YES)
                      { t.mark[n] = YES;
                        printnod;
                        n = t.llink[n];
                        l = n;
                      }
                  else
                      { printnod;
                        n = -t.rlink[n];
                        if (t.label[n] == parent) l = n;
                      }
                }
            else if (t.llink[n] == NULL && t.rlink[n] > 0) /* B --> RaB */
                { if (debug) printf ("%s", "***RaS n");
                  printnod;
                  t.mark[n] = YES;
                  n = t.rlink[n];
                  l = n;
                }
          }
    }

```

Figure 3.2.1: Binary Tree Constructor Algorithm
(Page 2 of 3)


```

        else if (t.llink[n] == NULL && t.rlink[n] < 0)
        {
            /* B --> Qa */
            if (debug) printf ("%s", "***Qa n");
            printnod;
            t.mark[n] = YES; n = -t.rlink[n];
            if (t.label[n] == parent) l = n;
        }
    } /* end of while on n */
} /* end of else */
if (linktype == 'L' && t.llink[c] == NULL)
{ if (debug) printf ("%s", "*** inserting leftchildn");
  t.rlink[indexin] = -c;
  t.llink[indexin] = NULL;
  t.label[indexin] = nodelabl;
  t.llink[c] = indexin;
  printf ("%c", nodelabl);
  return (SUCCESS);
}
else if (linktype == 'L' && t.label[l] == parent
        && t.llink[l] == NULL)
{ if (debug) printf ("%s", "*** inserting dis leftchildn");
  t.llink[l] = indexin;
  t.llink[indexin] = NULL;
  t.rlink[indexin] = -l;
  t.label[indexin] = nodelabl;
  printf ("%c", nodelabl);
  return (SUCCESS);
}
else if (linktype == 'R' && t.label[l] == parent
        && t.rlink[l] < 0)
{ if (debug) printf ("%s", "*** inserting rightchildn");
  t.rlink[indexin] = t.rlink[l];
  t.llink[indexin] = NULL;
  t.rlink[l] = indexin;
  t.label[indexin] = nodelabl;
  printf ("%c", nodelabl);
  return (SUCCESS);
}
else if (linktype == 'R' && t.rlink[c] == -HEAD)
{ if (debug) printf ("%s", "*** inserting rightchildn");
  t.rlink[indexin] = t.rlink[c];
  t.llink[indexin] = NULL;
  t.label[indexin] = nodelabl;
  t.rlink[c] = indexin;
  printf ("%c", nodelabl);
  return (SUCCESS);
}
else return (ERROR);
} /* end of consrtbt */

```

Figure 3.2.1: Binary Tree Constructor Algorithm
(Page 3 of 3)

For example, in a RTBT production with the following right-hand side,

$$\langle\langle \lambda a \quad T \rho a \quad T \rangle\rangle$$

$$[i] \quad [i]$$

the leftmost "T" is expanded (using one or more T-productions) to produce a terminal string before the rightmost "T" is expanded.

3.2.2 The Correspondence.

The following subsections address the elements of correspondence between the formal and programming language approaches to data structures. Terminal symbols, nonterminal symbols, data structure grammars and K-formulas, production selection and traversal order are discussed.

3.2.2.1 Terminal Symbols. In a K-formula, a node is referenced by its node label, and fields of a node are denoted by prefix operators on the node label, such as the link symbol " ρ " in the K-formula $\rho h \rho a \rho b h$. A programming language data structure declaration typically describes the format of a node using facilities such as the C-language "struct" (KR78); fields within a node are denoted by declarations of the variables which comprise the node.

3.2.2.2 Nonterminal Symbols. The appearance of a nonterminal symbol in a sentential form indicates that one or more additional nodes are to be generated in the

K-formula which represents the corresponding data structure. In an algorithm such as that in Figure 3.2.1, the corresponding factors are iteration to visit the next node and insertion of a node at a particular position in a structure.

3.2.2.3 Data Structure Grammars and K-Formulas. A data structure grammar defines the allowable sequences of terminal symbols, as declarations in a program define the allowable node formats. A K-formula defines a specific instance of a data structure and its traversal, which is represented in a program by the combination of a data structure declaration, some method (declaration, algorithm) of initializing the structure and a traversal algorithm.

3.2.2.4 Selection of a Production. When replacing the nonterminal "B" during the left-most derivation of a K-formula, selection of a production from a set of alternate B-productions corresponds with determining which set of logic to use in a programming language depiction of a constructor algorithm. In Figure 3.2.1, the relevant K-grammar A-productions are noted as comments in the logic which precedes the depth-first search, and the B-productions are noted as comments in the search and insertion portions of the algorithm. In essence, selection of a K-grammar production corresponds with "case" selection in a programming language algorithm.

3.2.2.5 Traversal Order. A K-grammar generates a K-formula which gives both the topology of a data structure and a particular traversal of the structure. In a constructor such as that of Figure 3.2.1, the traversal of a structure is distributed throughout the search portion of the algorithm. This algorithm was developed to correspond with the grammar RTBT; a shorter but equivalent algorithm not based on the grammar can be written to accomplish the same function.

3.2.3 Differences Between the Classical and Formal Approaches.

As seen above, a K-grammar can generate all instances of K-formulas which represent a certain traversal of a particular type of data structure. For example, the grammar RTBT generates K-formulas which represent an inorder traversal of a right-threaded binary tree. Thus, the traversal order is specified in the declaration of the structure. In a programming language approach, the declaration of a structure allocates storage and possibly also initializes the structure, but the traversal order must be specified separately by a procedure written using imperative statements of the language.

3.3 Operations on Data Structures.

This section contains a heuristic description of data structure operations. The concepts presented here are

formalized in the next section. The K-grammars of Section 3.1 specify the set of well-formed K-formulas which represent valid instances of the relevant data structures. To develop the background for formalizing operations on the data structures, consider the derivation sequence for a singly-linked list. The productions of the grammar SLL are repeated here for convenience.

$$S \rightarrow \rho h T \quad (1)$$

$$T \rightarrow \langle \langle \rho a \quad T \rangle \rangle \quad (2)$$

$$[i] \quad q$$

$$T \rightarrow h \quad (3)$$

A derivation of the list L1: $h \rightarrow a \rightarrow b \rightarrow c \rightarrow h$ uses the production sequence (1)(2)(2)(2)(3) and an appropriate mapping function; the corresponding K-formula is $\rho h \rho a \rho b \rho c h$.

Suppose one wishes to modify L1 to obtain the list:

$$L2: h \rightarrow a \rightarrow b \rightarrow d \rightarrow c \rightarrow h$$

The K-formula $\rho h \rho a \rho b \rho d \rho c h$ for L2 is derived by the production sequence (1)(2)(2)(2)(2)(3) and an appropriate mapping function. The insertion of "d" into L1 involves setting the " ρ " link at "d" to point to "c" (i.e., setting the " ρ " link at "d" to the previous value of the " ρ " link at "b"), and resetting the " ρ " link at "b" to point to the newly-inserted node "d". From the K-formula vantage point, the symbols " ρb " and " ρd " participate in the insertion.

The insertion process causes the following mapping on the K-formulas which represent the data structure.

$$\rho h \rho a \rho b \rho c h \rightarrow \rho h \rho a \rho b \rho d \rho c h$$

The insertion can also be examined by study of the derivation of the initial and revised K-formulas. A convenient graphical depiction of the derivation involves use of "parse trees" (see AU77). Each interior node of a parse tree is labeled by some nonterminal "X", and the children are labeled, from left to right, by the symbols of the right side of the production which replaced X in the derivation. The leaves of the parse tree, read from left to right, constitute a sentential form called the yield or frontier of the tree.

The K-formula mapping

$$\rho\eta\rho\alpha\rho\beta\rho\chi \rightarrow \rho\eta\rho\alpha\rho\beta\rho\delta\rho\chi$$

is depicted in parse tree form in Figure 3.3.1, where node symbols are shown as they would appear after application of the phrase indeterminate resolution mapping function.

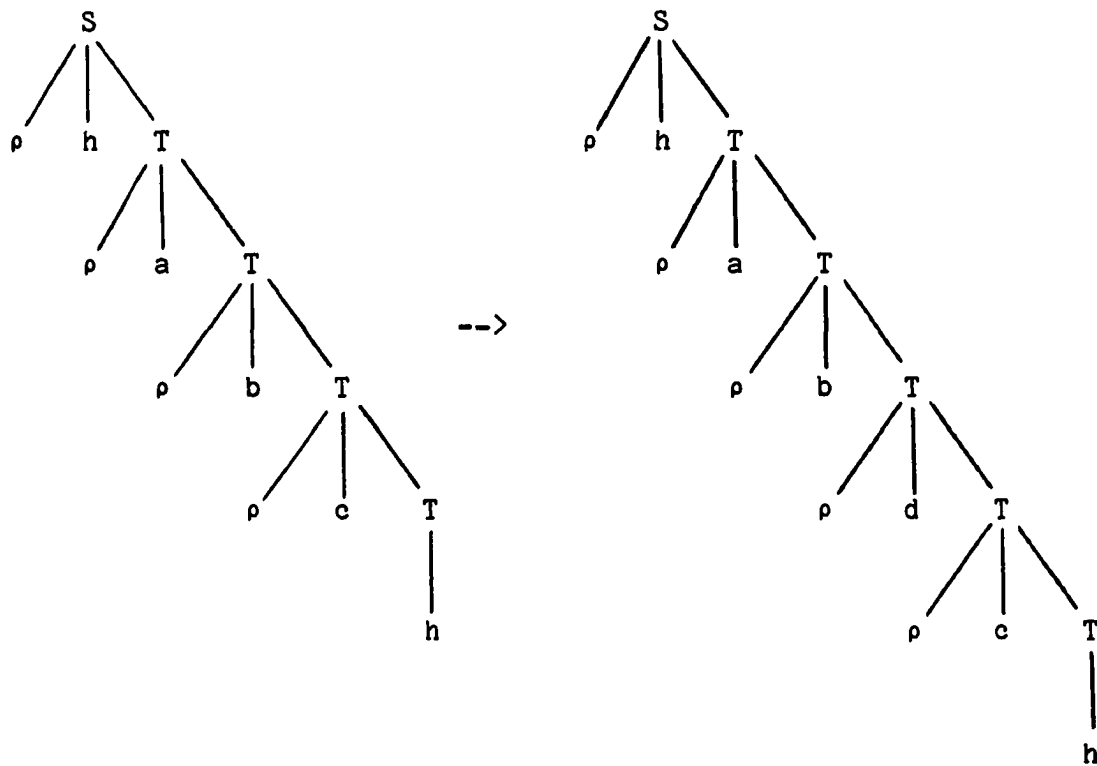


Figure 3.3.1: Derivation Trees for Singly-linked List Insertion.

The derivation portrayed in both parse trees is identical to the point where the nonterminal "T" which is a sibling of "ρ" and "b" is replaced by "ρdT" instead of "ρcT". That is, the insertion operation is characterized by the frontier

$$\rho h \rho a \rho b T \rightarrow \rho h \rho a \rho b \rho d T$$

At the point of interest in the (leftmost) derivation, all nonterminals preceding "ρbT" have been replaced by terminal symbols. The terminals "ρb" provide a point of reference for the insertion function. The nonterminal symbol "T" in both the left-side and the right-side of the operation ρbT → ρbρdT ultimately derives the terminal symbols ρch. In a sense, "T" represents a substructure of the portion of the structure which is of central interest in the transformation. In other structures, a substructure may itself be involved in a transformation.

Operations on Type-E linked data structures can thus be characterized by a transform which involves:

- (a) A point of reference at which the transformation occurs, termed the "reference node";
- (b) Node and link symbols which describe the operation performed; and
- (c) (If required) symbols which represent any substructures which are relevant to the transformation.

It is not necessary to show the entire K-formula to express

the change, but only enough node, link and substructure symbols to describe the relevant aspects of the transformation.

When inserting or deleting a node or substructure in a structure, note that an inserted node or substructure must be obtained from some source, and a deleted node or substructure must be returned to some destination. Consequently, a complete transform specification must really involve two structures, or two operations on a single structure. The complete transform for insertion of a free node into a SLL then becomes:

$$\rho f \rho x A \rightarrow \rho f A \Rightarrow \rho b B \rightarrow \rho b \rho x B$$

where:

- (a) "f" is the "header node" of a freelist;
- (b) "b" is the reference node in the singly-linked list;
- (c) "A" represents the rest of the freelist; and
- (d) "B" represents the portion of the singly-linked list which follows "b".

The node "x" is removed from the freelist, and is inserted after "b" in the singly-linked list of interest.

To preserve the integrity of the source and destination data structures being changed, the transform must be well-formed such that K-formulas which describe each structure can be derived from the appropriate data structure grammar. That is, given well-formed K-formulas which

correspond to the source and destination structures before the transform, the transform must guarantee that the K-formulas which correspond with the source and destination data structures after the transform are also well-formed. This assurance can be obtained if the components of the transform consist of well-formed symbol strings derivable from the right-hand-side (RHS) of a data structure production.

3.4 Formalisms for Operations on Data Structures.

The productions of a K-grammar define how a nonterminal symbol can be "rewritten" and thus precisely describe the pattern of symbols which can occur before and after a transformation of a structure. The sentential form sequences which generate the K-formulas which represent a structure before and after a transformation generally differ by one or two productions. Thus, a transform can be completely described by concentrating on the productions which express the changes. In the following, a primitive K-string consists of the right-hand side of a production, and a first-order K-string is a primitive K-string in which one or more nonterminals have been replaced by a valid production right-hand side. A second-order K-string is a first-order K-string in which one or more nonterminals have been replaced by a valid production right-hand side; higher order K-strings are similarly defined.

A transform which formally describes operations on

Type-E linked data structures is defined as follows.

transform ::= source ==> destination (3.4.1)

source ::= K-string --> K-string

destination ::= K-string --> K-string

K-string ::= primitive K-string or any valid
sentential form derived from a
primitive K-string, where:

- (1) the primitive K-string begins
with a link-node symbol pair;
and
- (2) any nonterminal symbols in
the K-string are replaced by
substructure symbols.

In the following, the terminology follows Aho and Ullman (AU72). (On this page and on the following page, ρ, λ, β denote K-strings and not specific link symbols.) For a grammar $G = (N, T, P, S)$, define a special type of string called a sentential form as follows:

a. S is a sentential form.

b. If $\rho B \lambda$ is a sentential form and $B \rightarrow \beta$ is in P, then $\rho \beta \lambda$ is also a sentential form.

A sentential form of G containing no nonterminal symbols is called a sentence of G.

Next, we define a relation " \Rightarrow " (read "directly derives") on $(N \cup T)^*$ such that if $\rho B \lambda$ is a string in $(N \cup T)^*$ and $B \rightarrow \beta$ is a production in P, then $\rho B \lambda \Rightarrow \rho \beta \lambda$. The symbols " \Rightarrow^+ " (read "derives in a nontrivial way") denote the transitive closure of " \Rightarrow ", and " \Rightarrow^* " (read "derives") denote the reflexive and transitive closure of " \Rightarrow ". The k-fold product of " \Rightarrow " is denoted by " $(k)\Rightarrow$ ";

indicates that β is derived from ρ by a derivation of length " k ".

That transforms as defined above are sufficient to describe all single-node and substructure insertion and deletion operations on the Type-E data structures addressed in this paper is shown as follows. For a K-formula (sentence) " $K1$ " which represents the pre-transform data structure and a K-formula " $K2$ " which represents the post-transform structure, these conditions hold.

- a. For some K-grammar $K = (N, T, P, S)$,
if $K \Rightarrow K1$, then $K \Rightarrow K2$.
- b. Suppose $K \Rightarrow \rho A \beta \Rightarrow K1$
and $K \Rightarrow \rho' A' \beta' \Rightarrow K2$,
where: $\rho, \beta, \rho', \beta'$ are in $(N \cup T)^*$
 A, A' are in N .

If $K1$ and $K2$ are the pre-transform and post-transform K-strings, respectively, then $\rho = \rho'$ and $\beta = \beta'$.

- c. For A and A' in Step b above,
if $A \Rightarrow \lambda$ and $A' \Rightarrow \lambda'$, then the data structure transformation

$$K1 \rightarrow K2$$

is precisely described by

$$\lambda \rightarrow \lambda'$$

where $\lambda = l a L$, $\lambda' = l' a L'$,

l and l' are link symbols,

"a" is the reference node symbol,

L, L' are in $(N \cup T)^*$.

From a strictly-theoretical perspective, any transform which generates valid K-formulas from valid input K-formulas performs a valid operation on a structure. Thus, any transform which meets Definition 3.4.1 may be considered valid. For a given grammar, one can depict all possible insertion operations for a destination structure by generating transforms in which the K-string to the right of the " $-->$ " is of order "n" or higher if the K-string to the left of the " $-->$ " is of order "n". For example, the destination portion of a transform given by

$$paT \rightarrow papbpcT$$

is a valid specification of insertion of the nodes labeled "b" and "c" into a singly-linked list. Similarly, the destination transform portion given by

$$\lambda a \rho \sim b \rho \sim a \rightarrow \lambda a \lambda c \rho \sim b \rho \sim c \rho \sim a$$

is a valid insertion of the node labeled "c" between nodes "a" and "b" in a right-threaded binary tree. Deletion operations can be similarly defined.

In normal application, however, a rather small subset of the set of all valid transforms for a given grammar is sufficient to represent the classical operations. For example, inserting a "leaf" node at the frontier of a right-threaded binary tree is a relatively common operation. From the grammar RTBT, inserting a left leaf child "b" of

the node "a" can be accomplished by either of the following transforms:

$\rho f \rho b S \rightarrow \rho f S \Rightarrow \rho a T \rightarrow \lambda \rho \sim b \rho a T$ ("a" had a right child)

$\rho f \rho b S \rightarrow \rho f S \Rightarrow \rho \sim a \rightarrow \lambda \rho \sim b \rho \sim a$ ("a" had no children)

These are the only transforms necessary to reflect single left leaf node insertion.

Relatively powerful operations can also be expressed using transforms. The following exchanges the subtrees of a right-threaded binary tree node "a".

$\lambda a S \rho a T \rightarrow \rho \sim a \Rightarrow \rho \sim a \rightarrow \lambda a T \rho a S$

The source and destination structures are one and the same in this instance.

Suppose one wishes to derive the transform used to effect a certain operation on a structure, given the relevant grammar and pre-transform and post-transform K-formulas. In the following procedure, portions of the pre-transform and post-transform K-strings are used to select the appropriate data structure productions.

a. Write the K-formula substrings (of terminal symbols) which describe the operation.

b. In the data structure grammar, find a symbol string in a primitive K-string in which:

- (1) the terminal symbols correspond with the pattern of terminal symbols which represent the reference node and the preceding link symbol (and possibly

other terminal symbols) in the pre-transform K-string, and

- (2) the nonterminal symbols can be expanded to yield the "unmatched" terminal symbols of the pre-transform K-string.

Note the nonterminal symbol which derives the selected primitive K-string.

c. Expand the nonterminal symbol from step b above to generate the desired post-transform K-formula substring, using a primitive, first-order or higher-order K-string as necessary. Note that the expansion must be based on a primitive K-string which begins with a link symbol followed by the reference node.

d. The transform consists of the following separated by "-->".

- (1) A pre-transform K-string consisting of a primitive or higher-order K-string in which any nonterminal symbols have been replaced by substructure symbols; this K-string was identified in step b above.
- (2) A post-transform K-string consisting of a primitive or higher-order K-string in which any nonterminal symbols have been replaced by substructure symbols; this K-string was identified in step c above.

Formally, the transform

$$t_{11} t_{12} t_{13} \dots t_{1m} \rightarrow t_{21} t_{22} t_{23} \dots t_{2n}$$

is specified as

$$\rho \rightarrow \beta$$

where:

- a. ρ is a primitive or higher-order K-string

$$l_1 (l_{[i]} | a_{[j]} | B'_{[k]})^*$$

derived from a nonterminal "A" in which

- (1) $l_1 = t_{11}$;
 (2) $a_1 = t_{12}$;
 (3) $t_{13} \dots t_{1m}$ are link terminal symbols

$l_{[i]}$, node terminal symbols $a_{[j]}$, or

can be derived from nonterminal

symbols $B_{[k]}$; and

- (4) the nonterminal symbols $B_{[k]}$ are

replaced by substructure symbols $B'_{[k]}$.

- b. β is a primitive or higher-order K-string

$$l'_1 (l_{[i]} | a_{[j]} | B'_{[k]})^*$$

derived from a nonterminal "A" in which

- (1) $l'_1 = t_{21}$;
 (2) $a_1 = t_{21}$;

(3) $t_{23} \dots t_{2n}$ are link terminal symbols

$l_{[i]}$, node terminal symbols $a_{[j]}$, or

can be derived from nonterminal

symbols $B_{[k]}$; and

(4) the nonterminal symbols $B_{[k]}$ are

replaced by substructure symbols $B'_{[k]}$.

Note that for an insertion operation, the post-transform K-string must be of equal or higher order than the pre-transform K-string, because the sentential form sequence used to derive the post-transform K-formula contains as many nonterminal expansions as does the sentential form sequence for the pre-transform K-string. Similarly, for a deletion operation, the pre-transform K-string must be of equal or higher order than the post-transform K-string.

As an example, consider an insertion operation for a structure associated with the grammar RTBT. The operation is described by the K-formula transform

$$\rho b \rho^{\sim} c \rightarrow \lambda b \rho^{\sim} d \rho b \rho^{\sim} c.$$

Referring to the grammar RTBT of Section 3.1, the following procedure is used to develop the destination transform.

a. Production (5) of RTBT matches the pattern of the pre-transform K-string when a nonterminal symbol B is used to derive $\rho^{\sim} c$. The transform leftpart consists of $\rho b S$, where the substructure symbol S replaces the nonterminal B .

b. Another B-production of the grammar RTBT, production (3), matches the pattern of the post-transform K-string when the leftmost nonterminal derives ρ^*d and the rightmost nonterminal is replaced by the substructure symbol S.

The destination portion of the transform is

$$\rho bS \rightarrow \lambda b \rho^* d \rho bS.$$

In this chapter, the correspondence of K-grammars and classical data structure generation has been examined, and a method of formally specifying operations on Type-E data structures using K-strings has been described. To illustrate these techniques, the next chapter gives several examples of data structure modifications.

CHAPTER FOUR

4. Application of Data Structure Transforms.

4.1 Method of Application.

Illustrations of the use of transforms on Type-E data structures are presented in this chapter. An interpreter, documented in Appendix 1, has been implemented to perform operations on the in situ structures. The interpreter operates on transforms which conform to the BNF presented in Section 3.4, but which transforms have been augmented to specify the types of the source and destination data structures. That is, the input to the interpreter is pseudo-programming language imperative statements which specify the transforms.

The interpreter of Appendix 1 operates on two data structures: a circular singly-linked list of unused nodes termed the "freelist" and the data structure which is transformed, termed the "target". Operations consist of removing a node from the freelist and inserting it into the target structure; removing a node from the target structure and inserting it into the freelist; and moving nodes or substructures about within the target structure. The interpreter can transform singly-linked circular lists, doubly-linked circular lists and right-threaded binary

trees. The nodes of these structures must be atomic; embedded substructures are not allowed in this implementation.

The in situ data structures are initialized at the beginning of the interpreter execution in the present implementation; a more complete implementation (see the discussion in Chapter 5) would include facilities for declaring a variety of data structures, as well as a method of naming specific instances of structures in the declarative statements and in the imperative transform statements. Before transform statements are executed, a syntax check is performed as described in the following section.

4.2 Syntax Checking Transform Statements.

Using the facilities of lex (LS75) and yacc (JS75), a capability to verify the syntax of a set of transform statements has been developed. The lexical analysis input specification is shown in Appendix 2, and the parser input specification is given in Appendix 3. Transform statements include designation of node labels, links and substructure labels.

The valid node labels include the lower case alphabetic characters a,b,c,d,e,g, and "i" through "z". The character "f" is reserved to denote the head of the freelist, and the character "h" denotes the head of the data structure being transformed. Left links are denoted by "L", right links by

"R" and threads by "Q". The uppercase characters "S" through "Z" may be used to denote substructures. Occasionally, the character "N" is used to denote a null (empty) substructure.

The syntax checking mechanism verifies that transform statements identify the types of the source and destination structures. (Valid types are "free" for the list of all free cells; "sll" for singly-linked lists; "dll" for doubly-linked lists; and "rtbt" for right-threaded binary trees.) Also verified is the link-node sequence, accepting only K-strings which are valid for the relevant structures. Note that substructure symbols are accepted in many of the transforms. Error messages are produced to indicate the following.

- o Failure to recognize the transform as valid.
- o Unmatched reference nodes in the source or destination portion of the transform.
- o Illegal identical node or substructure identifiers where uniqueness is required.
- o Unmatched node labels, where identical labels are required, as in the K-strings for certain right-threaded binary tree operations.

Syntax checking of transform statements is an important issue. Transforms which are derivable from the data structure grammar preserve the integrity of the in situ structure. The syntax checking of transform statements

allows the use of syntactic methods to assess the semantics of the K-transforms; invalid transforms can be rejected before use.

4.3 Interpreter Operation.

The interpreter is designed for interactive operation, accepting commands from a terminal user. The main components of the command sequences illustrated in Section 4.4 are transform statements which use the following syntax.

transform stype & dtype : slhs --> srhs ==> dlhs --> drhs ;

where:

stype specifies the type of the source data structure.

dtype specifies the type of the destination data structure.

slhs specifies the K-string of the source transform left-hand side.

srhs specifies the K-string of the source transform right-hand side.

dlhs specifies the K-string of the destination transform left-hand side.

drhs specifies the K-string of the destination transform right-hand side.

A transform statement specifies that a node or substructure moves from a source to a destination data structure.

After accepting a transform statement from the user, the interpreter displays the K-formula which corresponds

with the in situ data structure before the transform. The K-formula which corresponds with the transformed data structure is displayed at the conclusion of the transform operation. Other interpreter outputs include indications of whether a transform succeeded or failed, and certain error messages when appropriate.

Figure 4.3.1 gives the data flow of the interpreter. The interpreter proceeds by accepting a transform statement and certain control information from the user, and then performs the corresponding operations. The left-hand side of the source structure transform is processed first, followed by the right-hand side of the source transform. The destination data structure portion of the transform is then similarly processed.

Figure 4.3.2 gives the structure of the interpreter software. The "main" routine controls interaction with the user by displaying prompts, accepting inputs and displaying messages.

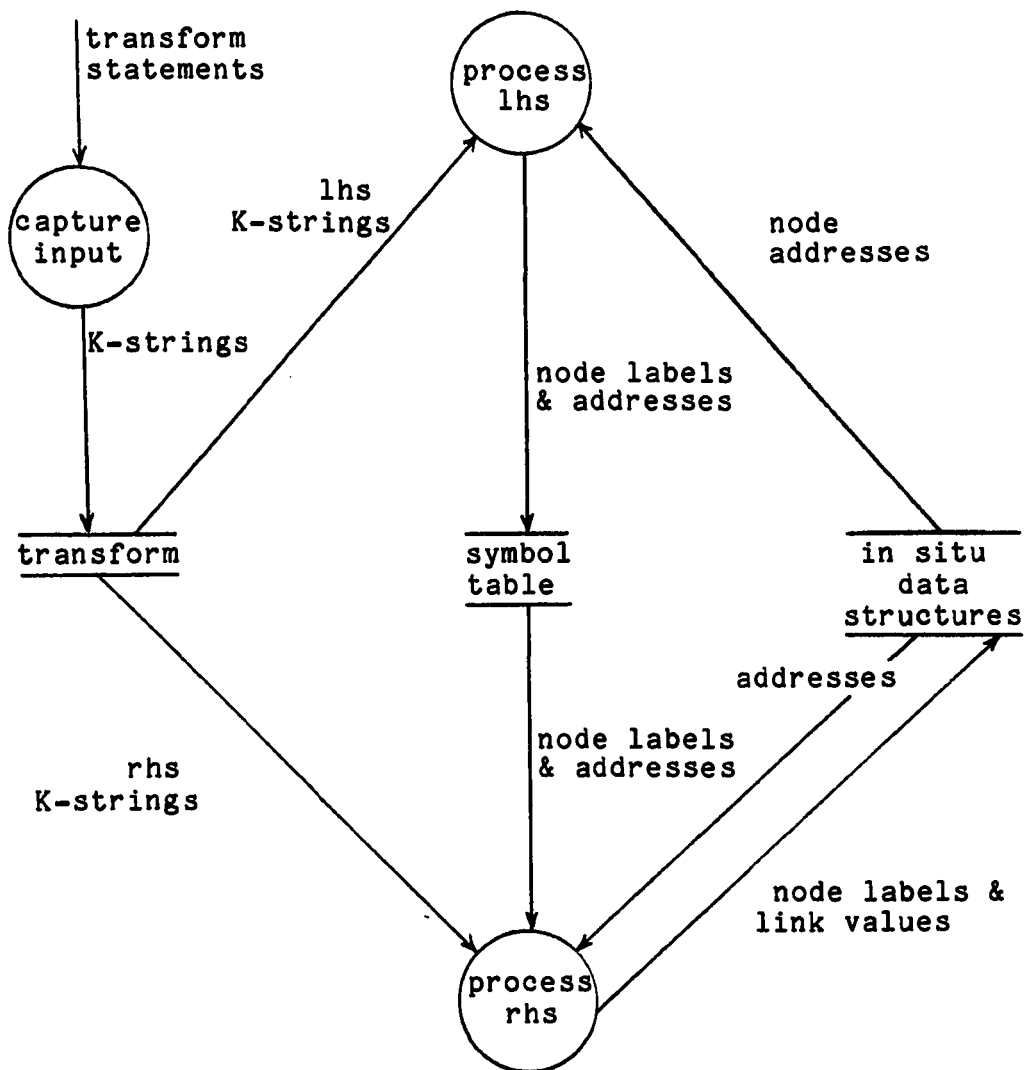


Figure 4.3.1: Interpreter Data Flow

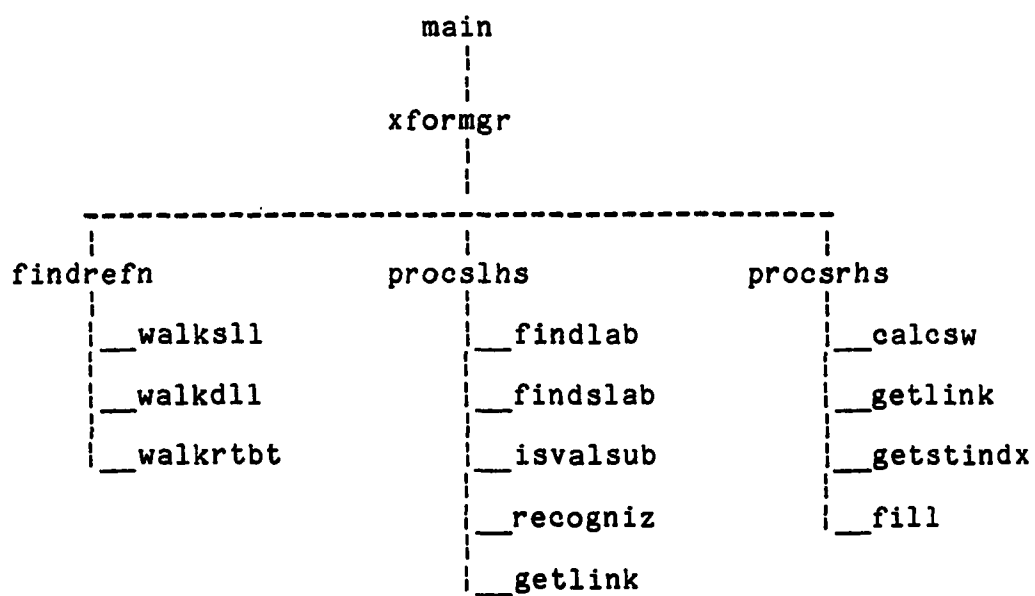


Figure 4.3.2: Interpreter Software Structure.

The procedure "xformgr" invokes a sequence of procedures which use the transform K-strings to effect changes to the in situ structure. "findrefn" calls the appropriate traversal algorithm (for the type of structure involved) to locate the reference node (see Section 3.3) in the in situ structure. The procedure "procslhs" builds a symbol table of node labels and the corresponding in situ locations, and substructure labels with the corresponding beginning and ending node locations. "procsrhs" uses the symbol table prepared by "procslhs" to modify the in situ structure, beginning at the reference node and successively setting links as prescribed by the appropriate right-hand side K-string. An important service routine is "recogniz", which traverses a structure to identify the last node in a substructure, given the first node in the substructure. A collection of other subroutines is used to perform services such as displaying the K-formula which corresponds with the in situ structure, and obtaining link values.

4.4 Examples of Interpreter Use.

This section contains illustrations of use of the interpreter. The first example involves manipulation of a doubly-linked list used in a communications message switching software application. The second example shows modifications of a right-threaded binary tree.

4.4.1 Message Queue Example. In a store-and-forward message switching system, messages awaiting delivery are typically serviced first-in first-out (FIFO) by message priority. Consider a system with three message priorities: high, medium and low. A practical way to effect FIFO by priority message delivery is to store message identifiers in a doubly-linked queue with the "oldest" high-priority message at the front of the queue, followed by the newer high-priority messages, followed by the "oldest" medium-priority message, followed by newer medium-priority messages, followed by the low-priority messages.

Desired operations include inserting messages at the rear of any of the queue priority sections and removing messages from the front of the queue. Occasionally, a message may be removed from any point in the interior of the queue. These operations are implemented by noting the labels of the queue header node "h", the newest member "t" of the high-priority section, the newest member "m" of the medium-priority section, and the newest member "l" of the low-priority section. The transform statements which manipulate the message queue are exemplified in the following.

a. Remove the oldest high-priority message and return the associated storage to the freelist.

```
transform dll & free : phpxSλxh --> phSh ==> pf --> pfox ;
```

In this case, the substructure "S" represents all of the queue except for the nodes "h" and "x".

b. Insert a new message into the medium-priority section, after obtaining the required storage from the freelist.

```
transform free & dll : pfpx --> pf ==> pmSλm --> pmpxSλxλm
;
```

Here, "S" represents the low-priority section of the queue.

c. Remove an arbitrary member "y" from the queue and return the associated storage to the freelist. (Note that the predecessor "c" of "y" must be known.)

```
transform dll & free : pcpySλyλc --> pcSλc ==> pf --> pfpy
;
```

"S" represents that portion of the queue which follows the node "y" before the deletion.

Figure 4.4.1 illustrates the input to and output of the interpreter when processing the three transforms given above. In the output image, note that the transform statement is printed, followed by the K-formulas which correspond with the queue before and after each operation. Because of equipment limitations, the characters "L" and "R" are used in place of "λ" and "ρ" in the K-formulas.

```

transform dll & free : RhRxSLxh --> RhSh ==> Rf --> RfRx ;
y
transform free & dll : RfRx --> Rf ==> RmSLm --> RmRxSLxLm ;
y
transform dll & free : RcRySLyLc --> RcSLc ==> Rf --> RfRy ;
n

```

Figure 4.4.1a: Message Queue Input Transforms

=====

```

transform dll & free : RhRxSLxh --> RhSh ==> Rf --> RfRx ;
dll before transform:  RhRaRtRcRdRmRiRlLhLlLiLmLdLcLtLah
transform succeeded
dll after  transform:  RhRtRcRdRmRiRlLhLlLiLmLdLcLth

** more transformations ?? (y or n):  y

transform free & dll : RfRx --> Rf ==> RmSLm --> RmRxSLxLm ;
dll before transform:  RhRtRcRdRmRiRlLhLlLiLmLdLcLth
transform succeeded
dll after  transform:  RhRtRcRdRmRaRiRlLhLlLiLaLmLdLcLth

** more transformations ?? (y or n):  y

transform dll & free : RcRySLyLc --> RcSLc ==> Rf --> RfRy ;
dll before transform:  RhRtRcRdRmRaRiRlLhLlLiLaLmLdLcLth
transform succeeded
dll after  transform:  RhRtRcRmRaRiRlLhLlLiLaLmLcLth

** more transformations ?? (y or n):  n

```

Figure 4.4.1b: Message Queue Interpreter Output

4.4.2 Height-Balanced Binary Tree Example. The height of a tree is defined to be the length of the longest path from the root to a leaf node. By minimizing the height of subtrees in a binary search tree, one can reduce the time required to locate a particular node below that required in an arbitrary binary tree organization. Horowitz and Sahni (HS76) describe an AVL tree which is height balanced such that the height of the left subtree of any given node differs from the height of the right subtree at that node by no more than one level. The difference between the length of the left subtree and right subtree, termed the balance factor, is stored at each node and is used to reorganize the tree when an insertion or deletion causes the balance factor at any node to exceed an absolute value of one.

The types of reorganizations required to rebalance a tree are termed rotations, and involve moving nodes and subtrees such that the height balance is restored while preserving the proper search key ordering. Because the rotations involve substructures (subtrees), the K-string transforms developed in this paper are convenient abstractions for expressing the required operations. Consider the "LL" rotation defined in Figure 4.4.2. The rotation involves:

- (1) Identifying the parent of the unbalanced node "a".

- (2) Identifying the subtrees of "b".
- (3) Identifying the right subtree of "a".
- (4) Relinking the tree such that the previous parent of "a" becomes the parent of "b"; linking "a" as the rightchild of "b"; and connecting the previous right subtree of "b" as the new left subtree of "a". The left subtree of "b" remains as such, and the right subtree of "a" also retains its original connectivity.

Figure 4.4.3 gives an instance of a right-threaded binary tree before, during and after height rebalancing. Before rebalancing, the node labeled "b" is the closest ancestor of the inserted node "m" for which the balance factor exceeds one in absolute value. Thus, rebalancing the left subtree of the node labeled "a" has the effect of rebalancing the entire tree. In Figure 4.4.3, the transforms referenced are those given in Figure 4.4.4.

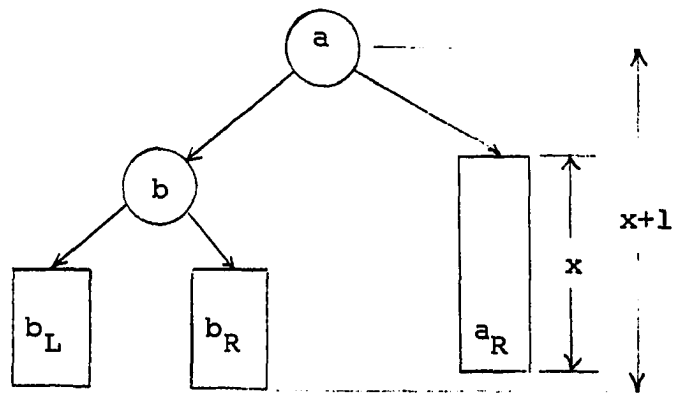


Figure 4.4.2a: Balanced AVL Tree

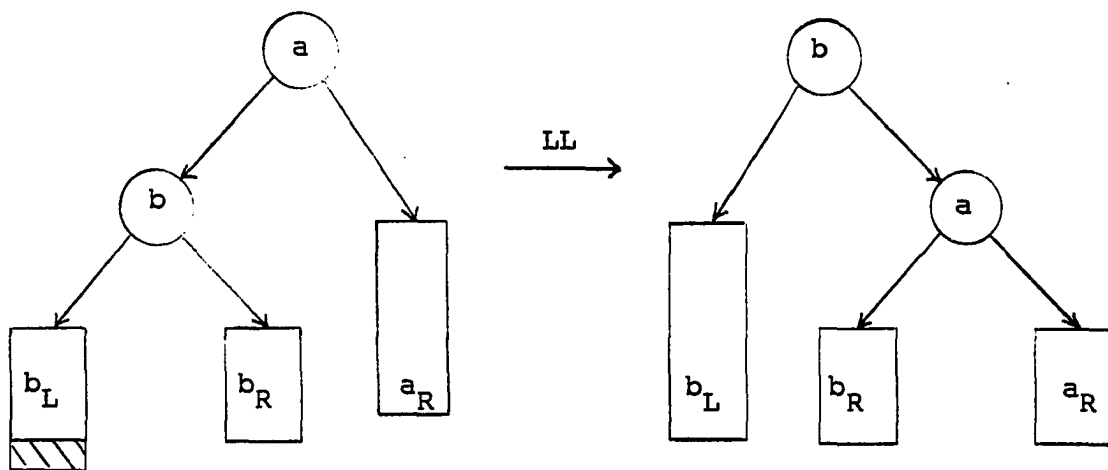


Figure 4.4.2b: AVL Tree "LL" Rotation

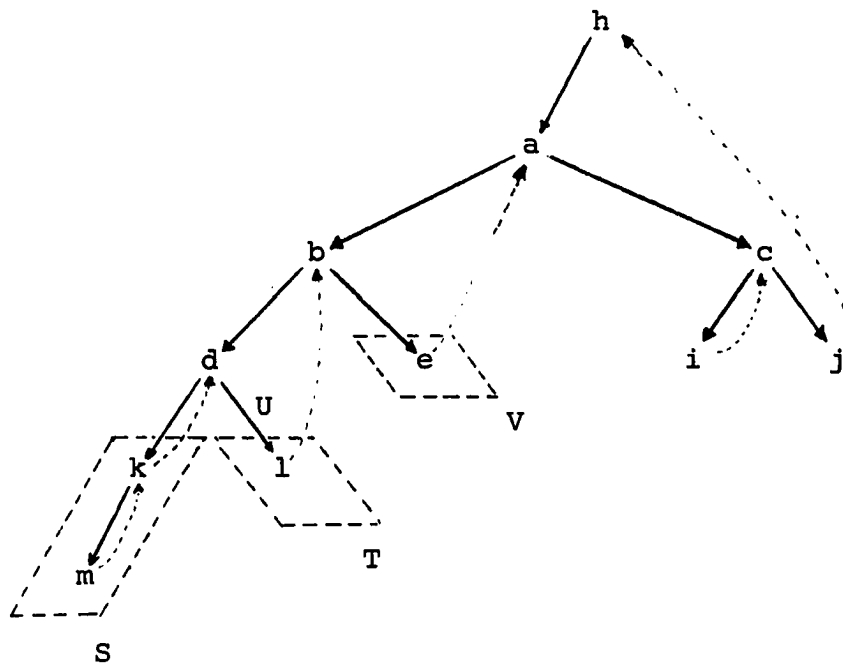


Figure 4.4.3a: Binary Tree Before Rebalancing

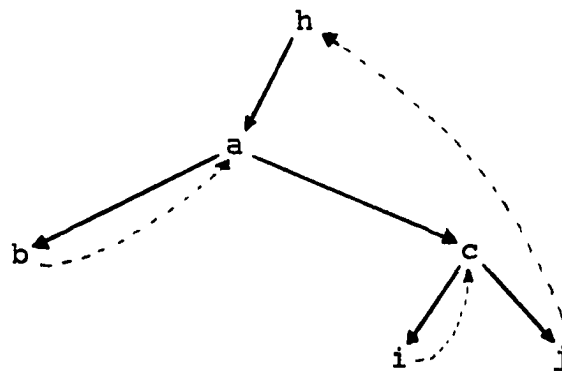


Figure 4.4.3b: Binary Tree After Transform 2

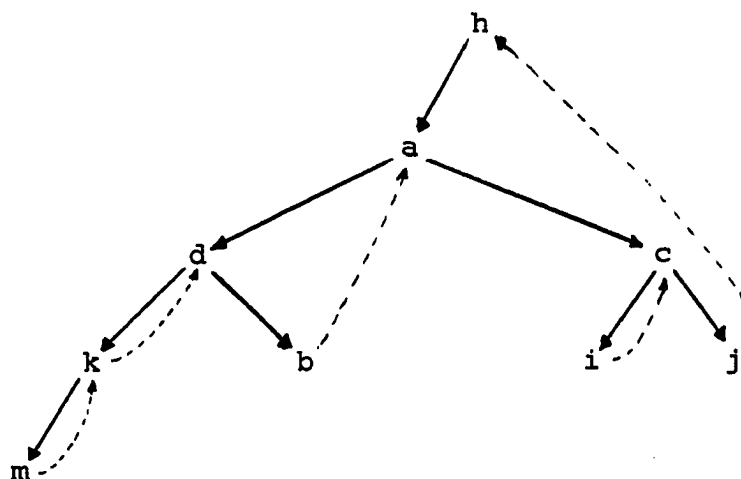


Figure 4.4.3c: Binary Tree After Transform 3

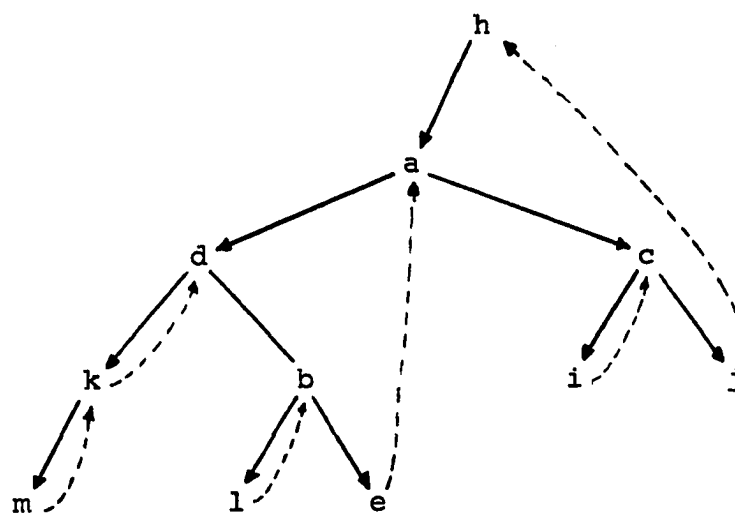


Figure 4.4.3d: Rebalanced Binary Tree

Figure 4.4.4a gives four transforms which rebalance the tree of Figure 4.4.3, using an "LL" rotation. Figure 4.4.4b shows the output of the interpreter when the transforms of Figure 4.4.4a are supplied as the input. The rotation proceeds as follows:

- (1) Transform statements "1" and "2" identify and disconnect the subtrees "S", "T", "U" and "V". Note that these actions are described in the "source" structure portion of the transform, and the "destination" structure portion of these transforms is essentially a null operation. The pseudo-substructure symbol "N" is used to set left links to a null value.
- (2) Transform statement "3" replaces "b" by "d" as the leftchild of "a", relinks substructure "S" as the left subtree of "d" and establishes "b" as the rightchild of "d". In the absence of a compiler implementation, substructure symbols are used where necessary to effect proper interpreter operation.
- (3) Transform statement "4" establishes substructures "T" and "V" as the left and right subtrees, respectively, of the node "b".

```

transform rtbt & free : LdSRdT --> LdNQd ==> Rf --> Rf ;
y
transform rtbt & free : LbURbV --> LbNQb ==> Rf --> Rf ;
y
transform rtbt & rtbt : LaQbRaX --> LaURaX ==> Qd --> LdSRdQb ;
y
transform free & rtbt : Rf --> Rf ==> Qb --> LbTRbV ;
n

```

Figure 4.4.4a: AVL "LL" Rotation Interpreter Input

=====

```

transform rtbt & free : LdSRdT --> LdNQd ==> Rf --> Rf ;
rtbt before transform:  LhLaLbLdLkQmQkRdQlRbQeRaLcQiRcQjh
transform succeeded
rtbt after transform:   LhLaLbQdRbQeRaLcQiRcQjh

** more transformations ?? (y or n):  y

transform rtbt & free : LbURbV --> LbNQb ==> Rf --> Rf ;
rtbt before transform:  LhLaLbQdRbQeRaLcQiRcQjh
transform succeeded
rtbt after transform:   LhLaQbRaLcQiRcQjh

** more transformations ?? (y or n):  y

transform rtbt & rtbt : LaQbRaX --> LaURaX ==> Qd --> LdSRdQb ;
rtbt before transform:  LhLaQbRaLcQiRcQjh
transform succeeded
rtbt after transform:   LhLaLdLkQmQkRdQbRaLcQiRcQjh

** more transformations ?? (y or n):  y

transform free & rtbt : Rf --> Rf ==> Qb --> LbTRbV ;
rtbt before transform:  LhLaLdLkQmQkRdQbRaLcQiRcQjh
transform succeeded
rtbt after transform:   LhLaLdLkQmQkRdLbQlRbQeRaLcQiRcQjh

** more transformations ?? (y or n):  n

```

Figure 4.4.4b: AVL "LL" Rotation Interpreter Output

The transforms of Figure 4.4.4 effect the "LL" AVL-tree rotation using K-strings derivable from the grammar RTBT of Chapter 3. Figure 4.4.5 gives another example of AVL tree rebalancing, this time using an "LR" rotation. Note that the left subtree of the node "p" is elided for clarity. The "RR" and RL" rotations (see HS76) can likewise be represented with K-transforms.

The AVL tree rotations are classically hard problems which typically perplex students who are being introduced to the structure. Applying the theory developed here, the rotations can be converted directly from pictorial depiction to transforms. The transforms allow the programmer to conceptualize the operations in terms of nodes, edges and substructures; whether the links are implemented using pointer variables or array indices is of little concern to the programmer. While the number of transform statements is approximately the same as the number of statements necessary using a modern programming language, the K-transforms can be syntactically evaluated to determine whether the integrity of the structure is preserved by each operation. Note that a right-threaded binary tree need not necessarily be used for the AVL tree representation. Because the right-threaded tree is Eulerian, it is convenient to use with regard to recognizability of the in situ structure.

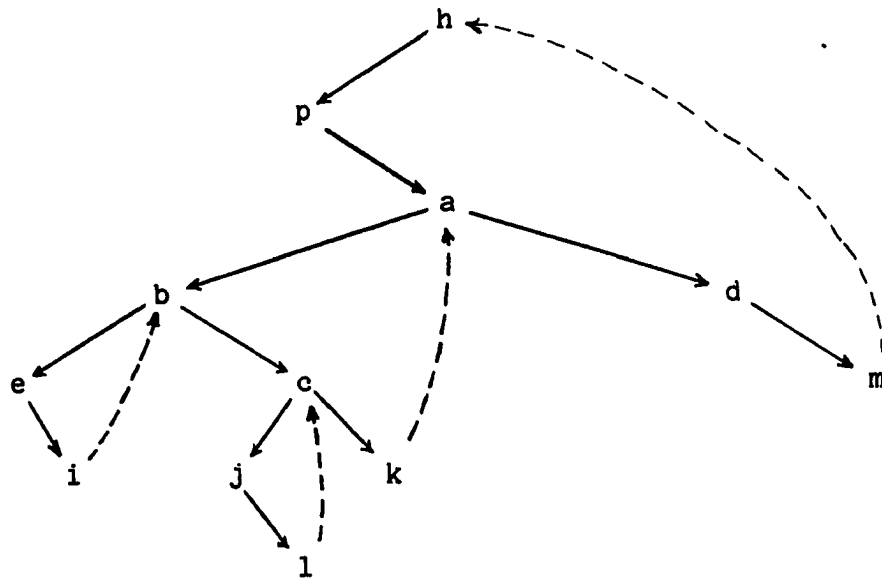


Figure 4.4.5a: AVL Tree Before "LR" Rebalancing

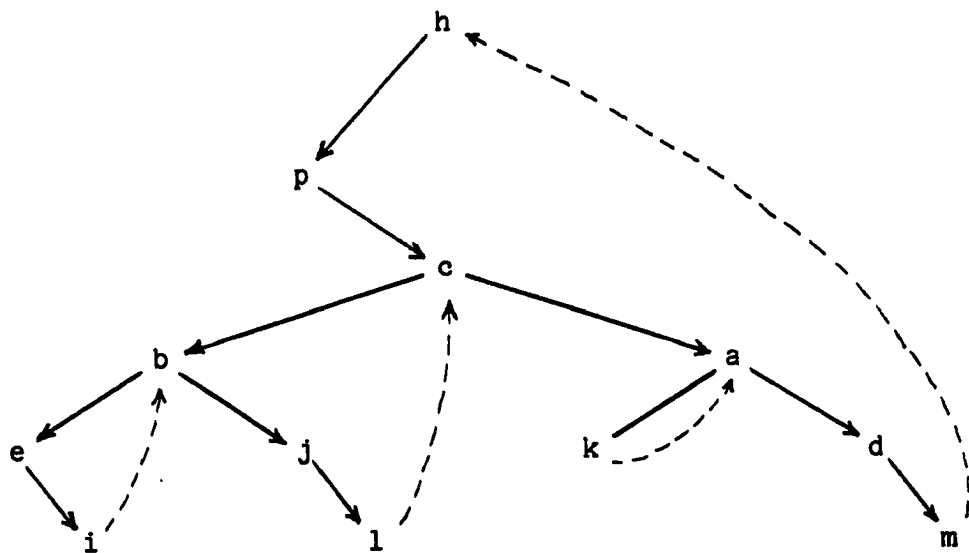


Figure 4.4.5b: AVL Tree After "LR" Rebalancing

```

transform rtbt & free : LbSRbLcTRcU --> LbNRbLcNQc ==> Rf --> Rf ;
y
transform rtbt & free : RbV --> Qb ==> Rf --> Rf ;
y
transform rtbt & rtbt : LaWRaX --> LaNQa ==> RpY --> RpV ;
y
transform free & rtbt : Rf --> Rf ==> Qc --> LcWRcY ;
y
transform free & rtbt : Rf --> Rf ==> LcQbRcQa --> LcLbSRbTRcLaURaX ;
n

```

Figure 4.4.5c: AVL "LR" Rotation Interpreter Input

```

transform rtbt & free : LbSRbLcTRcU --> LbNRbLcNQc ==> Rf --> Rf ;
rtbt before transform:  LhRpLaLbReQiRbLcRjQlRcQkRaRdQmh
transform succeeded
rtbt after transform:   LhRpLaRbQcRaRdQmh

** more transformations ?? (y or n):  y

transform rtbt & free : RbV --> Qb ==> Rf --> Rf ;
rtbt before transform:  LhRpLaRbQcRaRdQmh
transform succeeded
rtbt after transform:   LhRpLaQbRaRdQmh

** more transformations ?? (y or n):  y

transform rtbt & rtbt : LaWRaX --> LaNQa ==> RpY --> RpV ;
rtbt before transform:  LhRpLaQbRaRdQmh
transform succeeded
rtbt after transform:   LhRpQch

** more transformations ?? (y or n):  y

transform free & rtbt : Rf --> Rf ==> Qc --> LcWRcY ;
rtbt before transform:  LhRpQch
transform succeeded
rtbt after transform:   LhRpLcQbRcQah

** more transformations ?? (y or n):  y

transform free & rtbt : Rf --> Rf ==> LcQbRcQa --> LcLbSRbTRcLaURaX ;
rtbt before transform:  LhRpLcQbRcQah
transform succeeded
rtbt after transform:   LhRpLcLbReQiRbRjQlRcLaQkRaRdQmh

** more transformations ?? (y or n):  n

```

Figure 4.4.5d: AVL "LR" Rotation Interpreter Output

In this chapter, the viability of the K-transform methodology has been demonstrated using syntax checking tools and an interpreter. The final chapter contains discussion of work necessary for production implementation of the theory developed above.

CHAPTER FIVE

5. Summary, Future Work and An Evaluation.

5.1 Summary.

This paper described a method of formally specifying operations on the class of (Type-E) linked data structures which contain an Eulerian traversal. All valid operations on a Type-E structure can be derived from the associated data structure grammar, and all other operations fail to change the structure in a way which conforms to the grammar. To illustrate the syntactic specification of linked data structure operations, an interpreter was implemented.

5.2 Future Work.

For efficient use of the theory described herein in a production environment, it should be possible to use linked data structures in a manner similar to the way in which general purpose programming languages support scalar variables. In addition to the imperative transform statements described in this paper, additional statements are required to define instances of data structures and to initialize them for effective use.

Declaration statements should name a data structure and identify its type. Ideally, the type should be specified by including or referring to the data structure grammar.

Because the grammar includes a traversal, separate declarations should be used for distinct traversals. Each declaration should identify sets of valid link symbols, node symbols, and substructure symbols. Data structure initialization can be implicit in the declaration, an explicit part of the declaration, or performed by the user (programmer).

The transform statements of this paper must be modified to include the structure labels. In addition to the statements which move nodes from one structure to another, and move substructures about within a structure, other types of statements may also be added. For example, statements which access the data stored at a node are needed. Statements which search for a specific node in a data structure, or the nodes adjacent to a particular node, should be available. It should also be possible to remove a substructure from a structure, and retain the substructure for use in a subsequent transform statement.

A significant action necessary to achieving production use of the theory is that of implementing a program generator, or a full language and the associated compiler. In the former case, declaration and transform statements would be translated into program segments in an existing programming language, which would then be compiled and executed in the host language environment. The latter approach involves incorporating the data structure

declarative and imperative statements into a new language, and developing a compiler and the support environment for the language. By implementing the data structure transforms in a new language, facilities convenient to data structure manipulation can be effected. The compiler approach also offers the potential for optimization of generated code, and full-scale error checking. Use of a compiler may result in the ability of the optimizer to choose the most effective representation of a structure, based upon the data structure grammar.

While Type-E data structures are frequently found in databases, operating systems, communications and many other computer science applications, there are other important data structures which are not Type-E. Examples include unthreaded binary trees and arrays. A method of recognizing the syntactic representation of such structures would further increase the utility of the methods embodied in this paper. A hierarchy of data structures which parallels the Chomsky language hierarchy (AU72) may provide a classification useful in determining the required capabilities of the mechanisms which are needed to operate on the data structures.

5.3 An Evaluation.

To apply the present syntactic approach to linked data structures requires some appreciation of grammars, K-formulas and other formal methods in language and graph

theory. One may argue that the "average programmer" may encounter difficulty when attempting to apply the syntactic methods because of the theoretical background required. This argument is mitigated somewhat by recognizing that in a full implementation, a grammar which specifies a structure (and an associated traversal) need only be written once, and thereafter can be easily applied by any number of programmers. With syntax checking mechanisms semiautomatically generated from the grammar, transforms can be checked for validity before executing the associated code. These methods of preserving the integrity of the in situ structure seem to be ample reward for the investment required to learn to use the syntactic approach.

An advantage of the method described in this paper which goes beyond the correctness issue is that of providing the programmer with a powerful tool for manipulating data structures. Each statement in a program can effect a rather significant change in a pair of data structures. Thus, the programmer can dwell on the problem to be solved without having to be concerned with many of the details of how the in situ structure will actually be changed. The result is improved programmer productivity. Because syntactic expressions are abstract, the programmer need not be concerned with the representation of Type-E data structures. Consequently, syntactic methods contribute to the data independence property (DC77) which is important in database

management.

No advance in technology is effected without investment. Regarding the techniques described herein, the potential benefits appear to warrant the investment required to fully implement the methodology.

BIBLIOGRAPHY

- AU72 Aho, A.V. and Ullman, J.D. The Theory of Parsing, Translation, and Compiling. Vol. 1: Parsing. Prentice-Hall, Englewood Cliffs, N.J. 1972.
- AU77 Aho, A.V. and Ullman, J.D. Principles of Compiler Design. Addison-Wesley, Reading, Massachusetts, 1977.
- BA75 Bertziss, A.T. Data Structures Theory and Practice. 2nd Edition, Academic Press, New York, 1975.
- CM70 Crespi-Reghizzi, S. and Marpurgo, R. "A Language for Treating Graphs." Communications of the ACM, Vol. 13, No. 5 (May 1970) pp. 319-323.
- CN59 Chomsky, N. "On Certain Formal Properties of Grammars." Information and Control, Vol. 2, No. 2, 1959, pp. 137-167.
- CW71 Chen, W. Applied Graph Theory. American Elsevier Publishing Company, New York, 1971.
- DC77 Date, D. An Introduction to Database Systems. Addison-Wesley, Reading, Massachusetts, 1977.
- EJ71 Earley, J. "Toward and Understanding of Data Structures." Communications of the ACM, Vol. 10, No. 10 (Oct 1971), pp. 617-627.
- FA71 Fleck, A. "Towards a Theory of Data Structures." Journal of Computer and Systems Sciences, Vol. 5, 1971, pp. 475-488.
- GJ77 Guttag, J. "Abstract Data Types and the Development of Data Structures." Communications of the ACM, Vol. 20, No. 6, 1977, pp. 396-404.

- GY76 Guha, R. and Yeh, R. "A Formalization and Analysis of Simple List Structures." Applied Computation Theory: Analysis, Design, Modeling, Prentice-Hall, 1976.
- HS76 Horowitz, E. and Sahni, S. Fundamentals of Data Structures. Computer Science Press, Potomac, Maryland, 1976.
- JS75 Johnson, S. "Yacc: Yet Another Compiler-Compiler." Computing Science Technical Report No. 32, 1975. Bell Laboratories, Murray Hill, NJ.
- KR78 Kernighan, B.W. and Richie, D.M. The C Programming Language. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
- LS75 Lesk, M. and Schmidt, E. "Lex - A Lexical Analyzer Generator." Computing Science Technical Report No. 39, October 1975. Bell Laboratories, Murray Hill, New Jersey.
- PF71 Pratt, T. and Friedman, D. "A Language Extension for Graph Processing and Its Formal Semantics." Communications of the ACM, Vol. 14, No. 7 (July 1971), pp. 460-467.
- RA71 Rosenberg, A. "Data Graphs and Addressing Schemes." Journal of Computer and System Sciences, Vol. 5, 1971, pp. 193-238.
- SS74 Schneiderman, B. and Scheuermann, P. "Structured Data Structures." Communications of the ACM, Vol. 17, No. 10 (October 1974), pp. 566-574.
- ST78 Standish, T. "Data Structures - An Axiomatic Approach." Current Trends in Computing, Vol. 4, Prentice-Hall, 1978.
- TJ81 Thompson, J.C. Information Structures Lecture Notes, University of Oklahoma, Norman, Oklahoma, 1981.

APPENDIX ONE

Interpreter Documentation

```

/* listing of interpreter program    summer 1982    */
#define seq(A,B) strcmp(A,B) == 0    /* string equality */
#define NL n
#define NLQ "n"
#define BUFSIZE 60    /* buffer used to print K-formulas */
#define NULL 999
#define STRUCSIZ 14    /* number of nodes in "in situ" structure */
#define FOREVER 1
#define XFSIZE 16    /* transform components size */
#define YES 1
#define NO 0
#define EQUAL 0
#define ERROR -9
#define SUCCESS -1
#define NUMNODES 20    /* number of node symbols allowed */
#define NUMSTRUC 9    /* number of substructure symbols allowed */
/* */
static char label [STRUCSIZ+1] = "Ehabcdeijklmnf";
static int llink[STRUCSIZ]={NULL,2,3,5,7,9,NULL,NULL,11,NULL,NULL,NULL,NU:
static int rlink[STRUCSIZ]={NULL,NULL,4,6,8,10,-2,-4,-1,-5,-3,-9,13,12};
/*
static char label [STRUCSIZ + 1] = "Ehatcdmilknf";
static int llink[STRUCSIZ] = {NULL,8,1,2,3,4,5,6,7,NULL,NULL,NULL};
static int rlink[STRUCSIZ] = {NULL,2,3,4,5,6,7,8,1,10,11,9};
*/
static int info[STRUCSIZ] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13};
static char lhsl[XFSIZE]; /* transform K-strings */
static char lhsr[XFSIZE];
static char rhsr[XFSIZE];
static char rhsl[XFSIZE];
static char lhs[XFSIZE];
static char rhs[XFSIZE];
static int listctr = 0;
static int indexbeg = 1; /* index of beginning of structure */
static char erpre[] = "***--> error: "; /* er msg prefix */
static int nextnts = 0; /* index of next avail node table slot */
static int nextsts = 0; /* index of next avail substruct slot */
static int nextnode; /* index of node following xformed node */
static char validstr[NUMSTRUC] = "STUVWXYZ"; /* valid substru labels*/
static char lelem[3]; /* left element used in procrhs */
static char relem[3]; /* right etc */
static char llinksym = 'L'; /* symbol denoting left link */
static char rlinksym = 'R'; /* symbol denoting right link */

```



```

static char thredsym = 'Q';      /* symbol denoting thread */
static char nullsym = 'N';      /* symbol denoting null label */
char sstype[5];                /* source structure type */
char dstype[5];                /* dest structure type */
char stype[5];                 /* struct type used to recogniz */
int debug = 0;
int listnode = 0;              /* 1 ==> list nodes, 0 ==> do not */
struct nodes {                 /* node symbol table */
    char lab;
    int indx;
} nodetab[NUMNODES];
struct substruc {              /* substructure symbol table */
    char lab;                  /* label */
    int indxb;                 /* index of beg node */
    int indxe;                 /* index of end node */
    char link;                 /* link type at end node */
} structab[NUMSTRUC];
main()
{
    /* "main" accepts transform statements */
    int numread,retcode,whilesw, i;
    char debugans[4], reply[4];
    whilesw = YES;
    while (whilesw)
    {
        for (i = 0; i < NUMNODES; i++) nodetab[i].lab = ' ';
        /*
        for (i = 0; i < NUMSTRUC; i++) structab[i].lab = ' ';
        */
        nextnts = 0;
        /* nextsts = 0 special program version for AVL tree example */
        /* printf ("%s", "*** debug ?? (y or n): ");
        scanf ("%s", debugans);
        if (seq(debugans,"y") || seq(debugans,"yes"))
            listnode = 1;
        else listnode = 0; */
        liststru("main");
        while(FOREVER)
        { /*
            printf("%s%s", "transform stype & dtype : slhs --> srhs ",
                    "=> dlhs --> drhs ");
            */
            numread = scanf ("%s%s%s%s%s%s%s%s%s%s%s", sstype,
                             dtype, lhsl, rhsl, lhsr, rhsr);
            if (numread == 6)
            { printf("n");
              break;
            }
        } /* end of while */
        printf("%s%s%s%s%s%s%s%s%s%s", "transform ", sstype,
               " & ", dtype, " : ", lhsl, " --> ", rhsl, " ==> ",
               lhsr, " --> ", rhsr, " ;n");
        retcode = xformgr();
    }
}

```

```

if(retcode == SUCCESS)
    printf("transform succeededn");
else printf("transform failedn");
if (seq(sstype,"free")) printlab (dstype," after transform: ");
    else printlab (sstype," after transform: ");
printf("n** more transformations ?? (y or n): ");
scanf("%s", reply);
printf("%s%s", reply, "n");
if (seq(reply,"y") || seq(reply,"yes"))
    whilesw = YES;
else whilesw = NO;
} /* end of main */
/* end of while on whilesw */
/* */

xformgr()
{ /* transform manager -- sequences remaining procedures */
    int refnode, retcode;
    if (seq(sstype,"free")) printlab (dstype," before transform: ");
    else printlab (sstype," before transform: ");
    liststru("xformgr1");
    refnode = findrefn(sstype, lhsl[1]); /* find reference node */
    if (refnode == ERROR)
        return (ERROR);
    strcpy(stype,sstype);
    strcpy(lhs,lhs1);
    retcode = procslhs(refnode); /* process left lhs */
    liststru("xformgr2");
    if (retcode == ERROR)
        return(ERROR);
    if (lhsl[1] != rhs1[1])
        { printf("%s%s%s", erpre, "first node labels of left lhs ",
            "and rhs not identical n");
        }
    return(ERROR);
}
strcpy(stype,sstype);
strcpy(rhs,rhs1);
retcode = procsrhs(refnode); /* process left rhs */
liststru("xformgr3");
if (retcode == ERROR)
    return(ERROR);
refnode = findrefn(dstype, lhsr[1]); /* find right ref node */
if (refnode == ERROR)
    return(ERROR);
strcpy(stype, dstype);
strcpy(lhs, lhsr);
retcode = procslhs(refnode); /* process right lhs */
liststru("xformgr4");
if (retcode == ERROR)
    return(ERROR);
if (lhsr[1] != rhsr[1])
    { printf("%s%s%s", erpre, "first node labels of right lhs ",
        "and rhs not identical n");
    }
}

```

```

        return(ERROR);
    }
    strcpy(stype,dstype);
    strcpy(rhs, rhsr);
    retcode = procsrhs(refnode);          /* process right rhs */
    liststru("xformgr5");
    if (retcode == ERROR)
        return(ERROR);
    return(SUCCESS);
}
/* end of xformgr */
/* */
procsrhs(refnode) /* process right hand side (modifies structure) */
int refnode;      /* index of first node involved in xform */
{
    int cursor;          /* next rhs symbol */
    int lindx, rindx;    /* indices of lelem & relem sym */
    int stindx;          /* structure table index */
    int sw;
    if (debug)
        printf("%s", "-----> entered procsrhsn");
    cursor = 0;
    cursor = fill(cursor); /* begin at reference node; set */
    if (cursor == ERROR)    /* links as prescribed by the */
        return (ERROR);    /* right-hand side K-string */
    strcpy (lelem, relem);
    cursor = fill(cursor);
    if (cursor == ERROR)
        return (ERROR);
    while (relem[0] != '\0')
    { sw = calcs (lelem, relem); /* determine rhs symbol types */
      if (listnode)
          printf("%s%d%s", "*** switch value = ", sw, "n");
      switch(sw)
      {
          case 11: /* link-node, link-node */
              lindx = getindx(lelem[1]);
              rindx = getindx(relem[1]);
              if (lindx == ERROR || rindx == ERROR) return (ERROR);
              if (lelem[0] == llinksym)
                  llink[lindx] = rindx;
              else if (lelem[0] == rlinksym)
                  rlink[lindx] = rindx;
              else if (lelem[0] == thredsym)
                  rlink[lindx] = -rindx;
              else return (ERROR);
              break;
          /* */
          case 12: /* link-node, node */
              lindx = getindx(lelem[1]);
              rindx = getindx(relem[0]);
              if (lindx == ERROR || rindx == ERROR) return (ERROR);
              if (lelem[0] == llinksym)

```

```

        llink[lindx] = rindx;
    else if (lelem[0] == rlinksym)
        rlink[lindx] = rindx;
    else if (lelem[0] == thredsym)
        rlink[lindx] = -rindx;
    else return (ERROR);
    break;

/* */
case 13: /* link-node, substructure */
    lindx = getindx (lelem[1]);
    stindx = getstind(relem[0]);
    if (lindx == ERROR || stindx == ERROR) return (ERROR);
    rindx = structab[stindx].indx;
    if (lelem[0] == llinksym)
        llink[lindx] = rindx;
    else if (lelem[0] == rlinksym)
        rlink[lindx] = rindx;
    else if (lelem[0] == thredsym)
        rlink[lindx] = -rindx;
    else return (ERROR);
    break;

/* */
case 14: /* link-node, null */
    lindx = getindx (lelem[1]);
    if (lindx == ERROR) return (ERROR);
    if (lelem[0] == llinksym)
        llink[lindx] = NULL;
    else return (ERROR);
    break;

/* */
case 31: /* substructure, link-node */
    stindx = getstind (lelem[0]);
    rindx = getindx (relem[1]);
    if (stindx == ERROR || rindx == ERROR) return (ERROR);
    lindx = structab[stindx].indx;
    if (structab[stindx].link == llinksym)
        llink[lindx] = rindx;
    else if (structab[stindx].link == rlinksym)
        rlink[lindx] = rindx;
    else if (structab[stindx].link == thredsym)
        rlink[lindx] = -rindx;
    else return (ERROR);
    break;

/* */
case 32: /* substructure, node */
    stindx = getstind (lelem[0]);
    rindx = getindx (relem[0]);
    if (stindx == ERROR || rindx == ERROR) return (ERROR);
    lindx = structab[stindx].indx;
    if (structab[stindx].link == llinksym)
        llink[lindx] = rindx;
    else if (structab[stindx].link == rlinksym)

```

```

        rlink[lindx] = rindx;
    else if (structab[stindx].link == thredsymb)
        rlink[lindx] = -rindx;
    else return (ERROR);
    break;
case 41:                                /* null, link-node */
    /* no relinking required */
    break;
    /* */
default:
    printf("%s%s", erpre, "invalid lhs-rhs combination in procsrhs");
    return (ERROR);
}                                         /* end of switch on sw */
strcpy (lelem, relem);
cursor = fill(cursor);
if (cursor == ERROR)
    return (ERROR);
}                                         /* end of while */
if (nextnode == NULL || nextnode == -NULL)
    return (SUCCESS);
if (isvalsub (lelem[0]) == YES)
{ stindx = getstind (lelem[0]); /* link last substruct to rest */
  if (stindx == ERROR) return (ERROR);
  lindx = structab[stindx].indx;
  if (structab[stindx].link == llinksym)
      llink[lindx] = nextnode;
  else if (structab[stindx].link == rlinksym)
      rlink[lindx] = nextnode;
  else if (structab[stindx].link == thredsymb)
      rlink[lindx] = nextnode;
  else return (ERROR);
}
else
{ lindx = getindx (lelem[1]);
  if (lindx == ERROR) return (ERROR); /* link last node to rest */
  if (debug)
      printf("%s%d%c%s", "endrhs lindx lelem[1] ", lindx, lelem[1], "n");
  if (lelem[0] == llinksym)
      llink[lindx] == nextnode;
  else if (lelem[0] == rlinksym)
      rlink[lindx] = nextnode;
  else if (lelem[0] == thredsymb)
      rlink[lindx] = nextnode;
  else return (ERROR);
}                                         /* end of else */
return (SUCCESS);
}                                         /* end of procsrhs */
}
proclhs(refnode)
int refnode;                            /* index of first node involved in xform */
{ /* build symbol table of node & substructure labels & indices */
  int lhscur, strucur;                  /* lhs and structure cursors */
  int tf;                               /* thread flag: YES => getlink -> thread */

```

```

int retcode;
if (debug)
    printf("%s", "-----> entered procslhsn");
if ((retcode = findlab(label[refnode])) == NO)
{
    /* label not yet in table */
    nodetab[nextnts].lab = label[refnode];
    nodetab[nextnts].indx = refnode;
    ++nextnts;
}
printnod("procslhs1");
strucur = getlink(lhs[0], refnode);
if (strucur < 0)
{
    strucur = -strucur;
    tf = YES;
}
else tf = NO;
if (strucur == ERROR)
    return (ERROR);
lhscur = 2;
/* */
while (lhs[lhscur] != '0')
{
    if (debug)
        printf("%s%c%d%s", "***lhs, strucur = ", lhs[lhscur], strucur, "n");
    if (lhs[lhscur] == llinksym || lhs[lhscur] == rlinksym
        || lhs[lhscur] == thredsymb)
    {
        /* link node pair */
        if ((retcode = findlab(lhs[lhscur + 1])) == NO)
        {
            /* label not yet in table */
            nodetab[nextnts].lab = lhs[lhscur + 1];
            nodetab[nextnts].indx = strucur;
            ++nextnts;
        }
        strucur = getlink (lhs[lhscur], strucur);
        if (strucur < 0)
        {
            strucur = -strucur;
            tf = YES;
        }
        else tf = NO;
        lhscur = lhscur + 2;
    }
    else if ((retcode = isvalsub (lhs[lhscur])) == YES)
    {
        /* substructure */
        retcode = recogniz (styp, strucur, lhs[lhscur]);
        if (retcode == ERROR)
            return (ERROR);
        strucur = retcode;
        if (strucur < 0)
        {
            strucur = -strucur;
            tf = YES;
        }
        else tf = NO;
    }
}

```

```

        lhscur++;
    }
    else if (YES)
    {
        /* must be a node symbol */
        if ((retcode = findlab (lhs[lhscur])) == NO)
        {
            /* label not yet in table */
            nodetab[nextnts].lab = lhs[lhscur];
            nodetab[nextnts].indx = strucur;
            ++nextnts;
        }
        strucur = NULL;
        lhscur++;
    }
} /* end of while */

if (tf == YES)
    nextnode = -strucur;
else nextnode = strucur;
printnod("proclhs2");
return (SUCCESS);
} /* end of proclhs */
/* */
recogniz(type, begin, slabel) /* recognize substructure */
char type[]; /* structure type */
int begin; /* index of beginning node */
char slabel; /* label of substructure */
{
    int cursor; /* used to traverse structures */
    if (seq(type, "sll"))
        return (ERROR); /* sll substructures not allowed */
    if (seq(type, "dll"))
    {
        if (findslab (slabel) == YES)
        { printf("%s%c%s",erpre,slabel," already in dll struct tab n");
          return (ERROR);
        }
    }
    else
    { structab[nextsts].lab = slabel;
      structab[nextsts].indxb = begin;
      structab[nextsts].indxe = begin;
      structab[nextsts].link = llinksym;
      nextsts++;
      return (SUCCESS);
    }
} /* end of dll recognizer */

if (seq (type, "rtbt"))
{ if (findslab (slabel) == YES)
  { printf ("%s%c%sn",erpre,slabel," already in rtbt struc tab");
    return (ERROR);
  }
  else
  { cursor = begin;
    while (cursor != indexbeg)

```

```

{
    if (debug)
        printf("%s%d%s", "***recogniz rtbt cursor = ", cursor, "n");
    if (rlink[cursor] > 0)
        cursor = rlink[cursor];
    else if (rlink[cursor] < 0)
    { structab[nextsts].lab = slabel;
      structab[nextsts].indxb = begin;
      structab[nextsts].indxe = cursor;
      structab[nextsts].link = thredsymb;
      nextsts++;
      return (rlink[cursor]);
    }
} /* end of while on cursor */
return (ERROR);
} /* end of else */
} /* end of rtbt recognizer */
printf("%s%s%s", erpre, "recognizer not implem for ", type, "n");
return (ERROR);
} /* end of recogniz */
/* */
findrefn(strtype, rnodlabl) /* find reference node */
char strtype[5];
char rnodlabl;
{
    int rnodindx;
    if (debug)
        printf("%s", "-----> entered findrefn");
    if (seq(strtype, "free"))
        return (STRUCSIZ - 1);
    else if (seq(strtype, "sll"))
        rnodindx = walksll (indexbeg, rnodlabl);
    else if (seq(strtype, "dll"))
        rnodindx = walkdll (indexbeg, rnodlabl);
    else if (seq(strtype, "rtbt"))
        rnodindx = walkrtbt (indexbeg, rnodlabl);
    if (rnodindx == ERROR)
    { printf("%s%s%c%s", erpre, "lhs node ", rnodlabl, " not foundn");
      return (ERROR);
    }
    else return (rnodindx);
} /* end of findrefn */
/* */
liststru(callpgm) /* list the data structure */
char callpgm[9];
{
    int i;
    if (listnode == 0) return;
    ++listctr;
    printf("%s%d%s%s", "data structure listing ", listctr, " from ",
           callpgm, "n");
} /* return; */

```



```

for(i=0; i< STRUCSIZ; ++i)
    printf("%2c%4d%4d%4d%s", label[i], llink[i], rlink[i], info[i],
        "n");
} /* end of liststru */
/* */
printnod(callpgm) /* list nodes in nodetab */
char callpgm[10];
{
    int i;
    if (debug == 0) return;
    printf("%s%s%s", " listing of nodetab from ", callpgm, "n");
    for (i=0; i< NUMNODES; ++i)
        printf("%2c%4d%s", nodetab[i].lab, nodetab[i].indx, "n");
} /* end of printnod */
/* */
isvalsub(testlab) /* is testlab a valid substructure label */
char testlab;
{
    int i;
    for (i=0; i<NUMSTRUC-1; ++i)
    {
        if (testlab == validstr[i])
            return (YES);
    }
    return (NO);
} /* end of isvalsub */
getlink (linktype, nodeindx)
char linktype; /* type of link */
int nodeindx; /* index of node for which link type desired */
{
    if (linktype == llinksym)
        return (llink[nodeindx]);
    if (linktype == rlinksym)
        return (rlink[nodeindx]);
    if (linktype == thredsymb)
        return (rlink[nodeindx]);
    printf ("%s%s%s", erpre, "invalid link symbol ", linktype,
        " detected in getlinkn");
    return (ERROR);
} /* end of getlink */
/* */
walksll(begnode, rnodlabl)
int begnode; /* index of beginning node of structure */
char rnodlabl; /* label of node whose index to be found */
{ /* search singly linked list for rnodlabl */
    /* cursor used to traverse structure */
    int cursor;
    cursor = begnode;
    while (rlink[cursor] != begnode)
    { if (label[cursor] == rnodlabl)
        return (cursor);
        cursor = rlink[cursor];
    }
}

```

```

    if (label[cursor] == rnodlabl)
        return (cursor);
    else return (ERROR);
} /* end of walksl1 */
/* */
walkdll (begnode, rnodlabl) /* search dll for rnodlabl */
int begnode;
char rnodlabl;
{
    /* walksl1 will visit all dll nodes */
    int retcode;
    if ((retcode = walksl1 (begnode, rnodlabl)) == ERROR)
        return (ERROR);
    else return (retcode);
} /* end of walkdll */
/* */
walkrtbt (begnode, rnodlabl)
int begnode; /* index of beginning of structure */
char rnodlabl; /* label of node whose index is to be found */
{
    /* find a certain node in rtbt */
    int cursor; /* used to traverse structure */
    int tf; /* thread flag used to id thread path */
    cursor = begnode;
    if (label[cursor] == rnodlabl)
        return (cursor); /* head node */
    cursor = llink[cursor];
    if (cursor == NULL)
        return (ERROR); /* empty tree */
    tf = NO;
    while (cursor != begnode)
    { if (label[cursor] == rnodlabl)
        return (cursor);
      if (llink[cursor] != NULL)
      { if (tf == NO)
          cursor = llink[cursor];
        else if (rlink[cursor] > 0)
        { cursor = rlink[cursor];
          tf = NO;
        }
        else if (rlink[cursor] < 0)
        { cursor = -rlink[cursor];
          tf = YES;
        }
      }
      /* end if on label */
    else if (llink[cursor] == NULL)
    { if (tf == NO && rlink[cursor] > 0)
        cursor = rlink[cursor];
      else if (tf == NO && rlink[cursor] < 0)
      { cursor = -rlink[cursor];
        tf = YES;
      }
    }
    }
    if (cursor == NULL) return (ERROR);
}

```

```

    } /* end of while on cursor */
    return (ERROR); /* if rnodlabl not found */
} /* end of walkrtbt */
/* */
findlab (nodelab) /* see if nodelab is in nodetab */
char nodelab;
{
    int i;
    for (i=0; i<NUMNODES; i++)
        { if (nodetab[i].lab == nodelab)
            return (YES);
        }
    return (NO);
} /* end of findlab */
/* */
findslab (nodelab) /* see if nodelab is in structab */
char nodelab;
{
    int i;
    for (i = 0; i < NUMSTRUC; i++)
        { if (structab[i].lab == nodelab)
            return (YES);
        }
    return (NO);
} /* end of findslab */
/* */
fill(fillcur) /* fill relem from rhs */
int fillcur;
{
    int retval;
    if (rhs[fillcur] == nullsym)
        { relem[0] = rhs[fillcur];
          fillcur++;
          relem[1] = '\0';
          return (fillcur);
        }
    if (isvalsub(rhs[fillcur]) == YES)
        { relem[0] = rhs[fillcur];
          fillcur++;
          relem[1] = '\0';
          return (fillcur);
        }
    else if (rhs[fillcur] == llinksym || rhs[fillcur] == rlinksym
             || rhs[fillcur] == thredsymb)
        { relem[0] = rhs[fillcur];
          relem[1] = rhs[fillcur + 1];
          relem[2] = '\0';
          retval = fillcur + 2;
          return(retval);
        }
    else if (rhs[fillcur] >= 'a' && rhs[fillcur] <= 'z')
        { relem[0] = rhs[fillcur];

```

```

        relem[l] = '0';
        fillcur++;
        return (fillcur);
    }
    else if (rhs[fillcur] == '0')
    {
        relem[0] = '0';
        return (fillcur);
    }
    else return (ERROR);
}
/* end of fill */
/* */
/* */
calcs (lstr, rstr) /* calc switch val from lelem & relem */
char lstr[], rstr[]; /* left and right elements */
{
    int swval = 0;
    int retcode;
    if (debug == YES)
        printf("%s%s%s", " entered calcs ", lstr, rstr, "n");
    if ((retcode = isvalsub(lstr[0])) == YES)
        swval = 30;
    else if (lstr[0] == nullsym)
        swval = 40; /* 41 ==> null, link-node */
    else if (lstr[0] == llinksym || lstr[0] == rlinksym
             || lstr[0] == thredsym)
        swval = 10;
    if ((retcode = isvalsub(rstr[0])) == YES)
        swval = swval + 3;
    else if (rstr[0] == nullsym)
        swval = swval + 4; /* 14 ==> link-node, null */
    else if (rstr[0] == llinksym || rstr[0] == rlinksym
             || rstr[0] == thredsym)
        swval = swval + 1;
    else if (rstr[0] >= 'a' && rstr[0] <= 'z')
        swval = swval + 2;
    return (swval);
}
/* end of calcs */
/* */
/* */
getindx(nodesym) /* determine index of nodesym */
char nodesym;
{
    int i, j;
    for (i = 0; i < nextnts; i++)
    {
        if (nodesym == nodetab[i].lab)
        {
            j = nodetab[i].indx;
            return (j);
        }
    }
    /* end of if */
}
printf("%s%s", erpre, "node symbol ", nodesym,
        " not found in nodetab by getindxn");

```

```

    return (ERROR);
} /* end of getindx */
/* */
/* */
getstind (strucsym) /* determine index of substruct sym */
char strucsym;
{
    int i;
    for (i = 0; i < nextsts; i++)
        { if (strucsym == structab[i].lab)
            return (i);
        }
    printf("%s%s%s", erpre, "structure symbol ", strucsym,
        " not found in structab by getstindn");
    return (ERROR);
} /* end of getstind */
printlab (structyp, when) /* list node labels of structure */
char structyp[]; /* type of structure */
char when[]; /* print before or after transform */
{
    int i, indx;
    int tf; /* thread flag "ON" if last link was thread */
    char bufr[BUFSIZE]; /* print buffer */ for (i = 0; i < BUFSIZE; i++)
        bufr[i] = ' ';
    if (seq(structyp, "sll"))
    {
        i = 1;
        indx = indexbeg;
        while (rlink[indx] != indexbeg)
        {
            bufr[i] = rlinksym;
            bufr[i+1] = label[indx];
            i = i + 2;
            if (i > BUFSIZE) return (ERROR);
            indx = rlink[indx];
            if (indx < indexbeg || indx > STRUCSIZ)
                {printf("%s%s", erpre, "cannot print structure--invalid linkn");
                 return (ERROR);
                }
        }
        /* end of while */
        bufr[i] = rlinksym; /* last node */
        bufr[i+1] = label[indx];
        bufr[i+2] = label[indexbeg];
        printf ("%s%s%s", structyp, when, bufr, "n");
    } /* end of if */
    else if (seq(structyp, "dll"))
    {
        i = 1;
        indx = indexbeg;
        while (rlink[indx] != indexbeg)
        {
            bufr[i] = rlinksym;

```

```

    bufr[i+1] = label[indx];
    i = i + 2;
    if (i > BUFSIZE) return (ERROR);
    indx = rlink[indx];
    if (indx < indexbeg || indx > STRUCSIZ)
    { printf("%s%s", erpre, "cannot print structure-- invalid linkn");
      return (ERROR);
    }
} /* end of while */
bufr[i] = rlinksym;
bufr[i+1] = label[indx];
i = i + 2;
if (i > BUFSIZE) return (ERROR);
indx = indexbeg;
while (llink[indx] != indexbeg)
{
    bufr[i] = llinksym;
    bufr[i+1] = label[indx];
    i = i + 2;
    if (i > BUFSIZE) return (ERROR);
    indx = llink[indx];
    if (indx < indexbeg || indx > STRUCSIZ)
    { printf("%s%s", erpre, "cannot print structure-- invalid linkn");
      return (ERROR);
    }
}
bufr[i] = llinksym;
bufr[i+1] = label[indx];
bufr[i+2] = label[indexbeg];
printf("%s%s%s", structyp, when, bufr, "n");
return (SUCCESS);
} /* end of if on dll */
else if (seq (structyp, "rtbt"))
{
    i = 1;
    if (rlink[indexbeg] == -indexbeg && llink[indexbeg] == NULL)
    { bufr[i] = thredsymb;
      bufr[i+1] = label[indexbeg];
      bufr[i+2] = label[indexbeg];
      printf ("%s%s%s", "structure of type ", structyp, bufr);
      return (SUCCESS);
    }
    bufr[i] = llinksym;
    bufr[i+1] = label[indexbeg];
    i = i + 2;
    indx = llink[indexbeg];
    tf = NO;
    while (indx != indexbeg)
    {
        if (llink[indx] != NULL)
        { if (tf == NO)
          {

```

```

        bufr[i] = llinksym;
        bufr[i+1] = label[indx];
        indx = llink[indx];
        i = i + 2;
    }
    else if (rlink[indx] > 0)
    {
        bufr[i] = rlinksym;
        bufr[i+1] = label[indx];
        indx = rlink[indx];
        tf = NO;
        i = i + 2;
    }
    else if (rlink[indx] < 0)
    {
        bufr[i] = thredsym;
        bufr[i+1] = label[indx];
        indx = -rlink[indx];
        tf = YES;
        i = i + 2;
    }
}
/* end of if on llink */
else if (llink[indx] == NULL)
{ if (tf == NO && rlink[indx] > 0)
{
    bufr[i] = rlinksym;
    bufr[i+1] = label[indx];
    indx = rlink[indx];
    i = i + 2;
}
else if (tf == NO && rlink[indx] < 0)
{
    bufr[i] = thredsym;
    bufr[i+1] = label[indx];
    indx = -rlink[indx];
    i = i + 2;
    tf = YES;
}
}
/* end of else if on llink */
if (i > BUFSIZE) return (ERROR);
}
/* end of while */
bufr[i] = label[indexbeg];
printf ("%s%s%s", structyp, when, bufr, "n");
}
/* end of else if on rtbt */
else printf ("%s", erpre, "invalid structure type in printlabn");
}
/* end of printlab */

```

APPENDIX 2

"lex" Specification

```

%%
"&"      {return (AND);}
";"      {return (COL);}
";"      {return (0);}
"-->"    {return (ARO);}
"==>"    {return (FARO);}
f        { yylval = yytext[0] - 'a';
          return (FHEAD); }
h        { yylval = yytext[0] - 'a';
          return (HEAD); }
transform {return (TRANSFORM);}
free     {return (FREE);}
sll      {return (SLL);}
dll      {return (DLL);}
rtbt     {return (RTBT);}
[a-ei-zg] {yylval = yytext[0] - 'a';
           return (NLAB);}
R        {return (RLINK);}
L        {return (LLINK);}
Q        {return (THREAD);}
N        {return (NUL);}
[S-Z]    {yylval = yytext[0] - 'A';
           return (SLAB);}
[A-KM-P] {return (ERR);}
[ ]+     {;}
[n]+     {;}

```


APPENDIX 3

"yacc" Specification

```

%start stmt
%token LLINK RLINK THREAD NLAB SLAB HEAD FHEAD
%token TRANSFORM AND SLL DLL RTBT FREE
%token ARO FARO COL ERR
%token NUL
%%
stmt : TRANSFORM FREE AND SLL COL frees ARO frees FARO slls ARO slls
      { if (($6 != $8) || ($10 != $12))
        printf ("%sn", "*** warning - unmatched reference nodes");
      }
    | TRANSFORM SLL AND FREE COL slls ARO slls FARO frees ARO frees
      { if (($6 != $8) || ($10 != $12))
        printf ("%sn", "*** warning - unmatched reference nodes");
      }
    | TRANSFORM FREE AND DLL COL frees ARO frees FARO dlls ARO dlls
      { if (($6 != $8) || ($10 != $12))
        printf ("%sn", "*** warning - unmatched reference nodes");
      }
    | TRANSFORM DLL AND FREE COL dlls ARO dlls FARO frees ARO frees
      { if (($6 != $8) || ($10 != $12))
        printf ("%sn", "*** warning - unmatched reference nodes");
      }
    | TRANSFORM FREE AND RTBT COL frees ARO frees FARO rtbts ARO rtbts
      { if (($6 != $8) || ($10 != $12))
        printf ("%sn", "*** warning - unmatched reference nodes");
      }
    | TRANSFORM RTBT AND FREE COL rtbts ARO rtbts FARO frees ARO frees
      { if (($6 != $8) || ($10 != $12))
        printf ("%sn", "*** warning - unmatched reference nodes");
      }
    | TRANSFORM RTBT AND RTBT COL rtbts ARO rtbts FARO rtbts ARO rtbts
      { if (($6 != $8) || ($10 != $12))
        printf ("%sn", "*** warning - unmatched reference nodes");
      }
    | TRANSFORM RTBT AND FREE COL rtbts ARO rtbtsn FARO frees ARO frees
      { if (($6 != $8) || ($10 != $12))
        printf ("%sn", "*** warning - unmatched reference nodes");
      }
    | TRANSFORM RTBT AND RTBT COL rtbts ARO rtbtsn FARO rtbts ARO rtbts
      { if (($6 != $8) || ($10 != $12))
        printf ("%sn", "*** warning - unmatched reference nodes");
      }

```

```

    }
;
frees : RLINK FHEAD
    { $$ = $2 ; }
    | RLINK FHEAD RLINK NLAB
    { $$ = $2 ; }
;
slls : RLINK NLAB
    { $$ = $2 ; }
    | RLINK NLAB RLINK NLAB
    { if ($2 == $4)
      printf("%sn", "***warning -- illegal identical node labels");
      $$ = $2;
    }
    | RLINK HEAD
    { $$ = $2 ; }
    | RLINK HEAD RLINK NLAB
    { $$ = $2; }
;
dlls : RLINK HEAD SLAB HEAD
    { $$ = $2; }
    | RLINK HEAD RLINK NLAB SLAB LLINK NLAB HEAD
    { if ($4 != $7)
      printf ("%sn", "*** warning - unmatched node labels");
      $$ = $2;
    }
    | RLINK NLAB RLINK NLAB SLAB LLINK NLAB LLINK NLAB
    { if (($2 != $9) || ($4 != $7))
      printf ("%sn", "*** warning - unmatched node labels");
      $$ = $2;
    }
    | RLINK NLAB SLAB LLINK NLAB
    { if ($2 != $5)
      printf ("%sn", "*** warning - unmatched node labels");
      $$ = $2;
    }
    | RLINK NLAB LLINK HEAD LLINK NLAB
    { if ($2 != $6)
      printf ("%sn", "*** warning - unmatched node labels");
      $$ = $2;
    }
    | RLINK NLAB RLINK NLAB LLINK HEAD LLINK NLAB LLINK NLAB
    { if (($2 != $10) || ($4 != $8))
      printf ("%sn", "*** warning - unmatched node labels");
      $$ = $2;
    }
    }
;
rtbts : THREAD HEAD HEAD
    { $$ = $2; }
    | LLINK HEAD SLAB HEAD
    { $$ = $2; }
    | THREAD NLAB

```

```

        { $$ = $2; }
| RLINK NLAB SLAB
  { $$ = $2; }
| LLINK NLAB SLAB THREAD NLAB
  { $$ = $2;
    if ($2 != $5)
      printf ("%sn", "***warning-unmatched node labels");
  }
| LLINK NLAB SLAB RLINK NLAB SLAB
  { $$ = $2;
    if ($2 != $5)
      printf ("%sn", "***warning-unmatched node labels");
    if ($3 == $6)
      printf ("%s%s", "warning-identical substructure ",
                  "labels");
  }
| RLINK NLAB THREAD NLAB
  { $$ = $2;
    if ($2 == $4)
      printf ("%sn", "***warning-illegal identical node labels");
  }
| LLINK NLAB THREAD NLAB THREAD NLAB
  { $$ = $2;
    if ($2 == $4 || $4 == $6)
      printf ("%sn", "***warning-illegal identical node labels");
    if ($2 != $6)
      printf ("%sn", "***warning-unmatched node labels");
  }
| LLINK NLAB THREAD NLAB RLINK NLAB SLAB
  { $$ = $2;
    if ($2 == $4 || $4 == $6)
      printf ("%sn", "***warning-illegal identical node labels");
    if ($2 != $6)
      printf ("%sn", "***warning-unmatched node labels");
  }
| LLINK NLAB THREAD NLAB RLINK NLAB THREAD NLAB
  { $$ = $2;
    if ($2 == $4 || $4 == $6 || $6 == $8
        || $2 == $8 || $4 == $8)
      printf ("%sn", "***warning-illegal identical node labels");
    if ($2 != $6)
      printf ("%sn", "***warning-unmatched node labels");
  }
| LLINK NLAB SLAB RLINK NLAB THREAD NLAB
  { $$ = $2;
    if ($2 == $7 || $5 == $7)
      printf ("%sn", "***warning-illegal identical node labels");
    if ($2 != $5)
      printf ("%sn", "***warning-unmatched node labels");
  }
| LLINK HEAD THREAD NLAB HEAD
  { $$ = $2; }

```

```

;
rtbtsn : LLINK NLAB NUL THREAD NLAB
        { $$ = $2;
          if ($2 != $5)
            printf ("%sn", "***warning-unmatched node labels");
        }
| LLINK NLAB NUL RLINK NLAB SLAB
        { $$ = $2;
          if ($2 != $5)
            printf ("%sn", "***warning-unmatched node labels");
        }
| LLINK NLAB NUL RLINK NLAB THREAD NLAB
        { $$ = $2;
          if ($2 == $7 || $5 == $7)
            printf ("%sn", "***warning-illegal identical node labels");
          if ($2 != $5)
            printf ("%sn", "***warning-unmatched node labels");
        }
| LLINK HEAD NUL THREAD HEAD HEAD
        { $$ = $2; }
;
%%
# include "lex.yy.c"

```