

Dines Bjørner

June 13, 1972

RJ 1053

Yorktown Heights, New York

San Jose, California

Zurich, Switzerland

Copies may be requested from:
IBM Thomas J. Watson Research Center
Post Office Box 218
Yorktown Heights, New York 10598

FINITE STATE TREE COMPUTATIONS

(PART I)

Dines Bjørner
IBM Research Division
San Jose, California

RJ 1053 (#17598)
June 13, 1972
Computer Sciences

THE HISTORY OF THE

OF THE

THE HISTORY OF THE

ABSTRACT

We generalize the notion of transitions, fstt's, of a finite state tree automaton and of finite state tree transformations, FSTTs. We then define the concept of finite state tree transformation evaluators, FSTEs, and construct several such evaluators in terms of FSTTs. We then go on to define context-free tree programming language systems and give synthesis algorithms which apply to a program in such a language and yields a realization of this program in terms of a single, and simple, tree walking finite state machine. The FSTEs provide, in the context of tree transformations, for computation rules elsewhere known either as call-by-value, call-by-name, or as leftmost-innermost-, leftmost-outermost-, Kleene-, standard- and normal- substitution rules [Ref. 10]. It should be noted that we can formulate such rules as tree transformation systems themselves; this being possible because of the absence of variables in our system. A tree programming language is thus a variable-free programming language. We can therefore relate (in PART II) the systems investigated to the classification of programming languages recently proposed by Backus [Ref. 2]. We regard our investigation presented here as an attempt to provide further theoretical and implementational support for such languages. In the appendix we give extensive examples of actual tree programming language programs. These denote the precise intensional semantics of LISP [Ref. 12], AE/ISWIM [Refs. 6-7] and RED/2 [Ref. 2].

CONTENTS

PAGE:	SECTION/SUBJECT:
i1	ABSTRACT
100	REFERENCES
126	NOTATION
i3	CONTENT
1	1. INTRODUCTION
1	1.1 Motivation, Problem and Justification
1	1.1.1 Motivation
1	1.1.2 Problem
1	1.1.3 Justification
4	1.2 A Theory of Process Description
5	1.3 Relation to other Work
5	1.4 Linguistic- and Programming Language Semantics
6	1.5 Rigorous Semantics
11	1.6 Disclaimer
12	1.7 Organization of Part I
15	2. DEFINITIONS
16	2.1 Alphabets
16	2.2 Trees
19	2.3 Simple Operations on Trees
29	3. TREE TRANSFORMATIONS
29	3.1 Finite State Tree Transitions
30	3.2 Applicability
31	3.3 Assignment

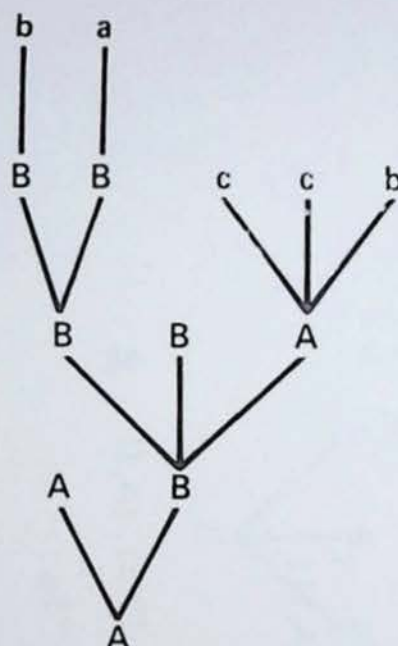
33	3.4 Substitution
33	3.5 Direct Generation
34	3.6 Finite State Tree Transformation
34	3.7 Proper FSTT's
35	3.8 Tree Computation
35	3.9 Tree Mapping
37	3.10 - alternative to 3.8
38	3.11 - discussion, on the presence of f-Operators
39	4. TREE EVALUATIONS
39	4.1 Universal f-Inspection Schemes
40	4.2 Finite State Tree Evaluator, FSTE
41	Notation
42	4.2.1 Note on Completion
43	4.3 Example: Canonical Outermost Evaluator, $\underline{Vco}$
45	4.4 Example: Normal Evaluator, $\underline{Vn}$
47	4.5 Example: Leftmost Innermost Evaluator, $\underline{Vli}$
49	5. DETERMINATE EVALUATIONS
49	5.1 Deterministic FSTT
50	5.2 Reduced FSTT
51	5.3 Partial and Total FSTT
52	5.4 Determinate, Serial and Free Computations
53	5.5 Second version of $\underline{Vco}$, Example
54	5.6 Second version of $\underline{Vn}$, Example
56	5.7 Second version of $\underline{Vli}$, Example
57	5.8 Completion
59	6. TREE AUTOMATA
59	6.1 Review of Tree Automata

64	6.2 Discussion
67	7. CONTEXT FREE TREE PROGRAMMING LANGUAGES
67	7.1 Languages and Meaning
68	7.2 Tree Programming
70	7.3 Finite State Tree Programming Systems, Θ
70	7.4 The Syntax of Θ Programs
70	7.5 The Semantics of Θ Programs
73	7.6 Context-free Θ Systems, Θ'
74	7.7 Context-free fstt
75	7.8 Term Tree fstt's
76	7.9 Discussion
79	8. SERIAL TREE MACHINES
80	8.1 Informal Exposé - and some definitions
83	8.2 The Algorithm
89	8.2.1 Production Partitioning
91	8.2.2 Index Set Update
92	8.2.3 Path Construction
94	8.2.4 Machine Definition
95	8.2.5 Machine Construction
97	Example
98	CONCLUSION, part I
99	ACKNOWLEDGEMENT
102	APPENDICES
102	<u>I</u> : The pure LISP 1.5 FSTT
105	<u>II</u> : The Landin AE 'S.E.C.D' FSTT
107	<u>III</u> : The RED/2 FSTT
118	<u>IV</u> : 30 Tree Diagrams for the V_n Evaluator
125	<u>V</u> : The RED/2 Serial Tree Machine

1. Introduction	1
2. Objectives and Scope	2
3. Literature Review	3
4. Methodology	4
5. Data Collection	5
6. Results and Discussion	6
7. Conclusion	7
8. References	8
9. Appendix	9
10. Glossary	10
11. Bibliography	11
12. Index	12
13. List of Figures	13
14. List of Tables	14
15. Acknowledgements	15
16. Declaration	16
17. Certificate	17
18. Curriculum Vitae	18
19. Statement of Work	19
20. Summary	20

FIG. 2.1

tree: t_1



E-expression

$A(A, B(B(B(b), B(a)), B, A(c, c, b)))$

$E = U = U_0 = \{A, B, a, b, c\}$

$U_1 = \{B\}$

$U_2 = U_3 = \{A, B\}$

yield $[t_1] = A b a B c c b$

bud $[t_1] = \underline{U} = \text{leaf}(t_1)$

height $[t_1] = 4$

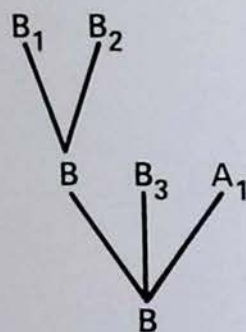
$A B B B a \in \text{path}[t_1]$

prefix $[t_1] = \{A, A(A, B), A(A(B, B, A)), \dots\}$

suffix $[t_1, A(A(B, B, A))] = \{B(B(b), B(a)), A(c, c, b)\}$

FIG. 2.2

tree: t_2



$B(B([B,1], [B,2]), [B,3], [A,1])$

$\underline{U} = \underline{U}_0 = \{A, B\};$

$\underline{U}_2 = \{B\} = \underline{U}_3; \underline{U}_1 = \emptyset$

$\underline{Z} = \underline{U} \times \underline{I};$ i.e. $[B,1], [B,2], [B,3], [A,1] \in \underline{Z}$

$\underline{n-p} [t_2] = B(B(B, B), B, A) = t_3$

$t_3 \in \underline{\text{interior}} [t_1]$

FIG. 3.1

fstt:

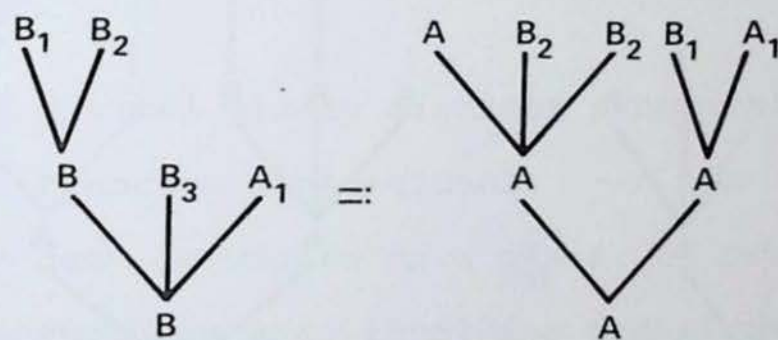
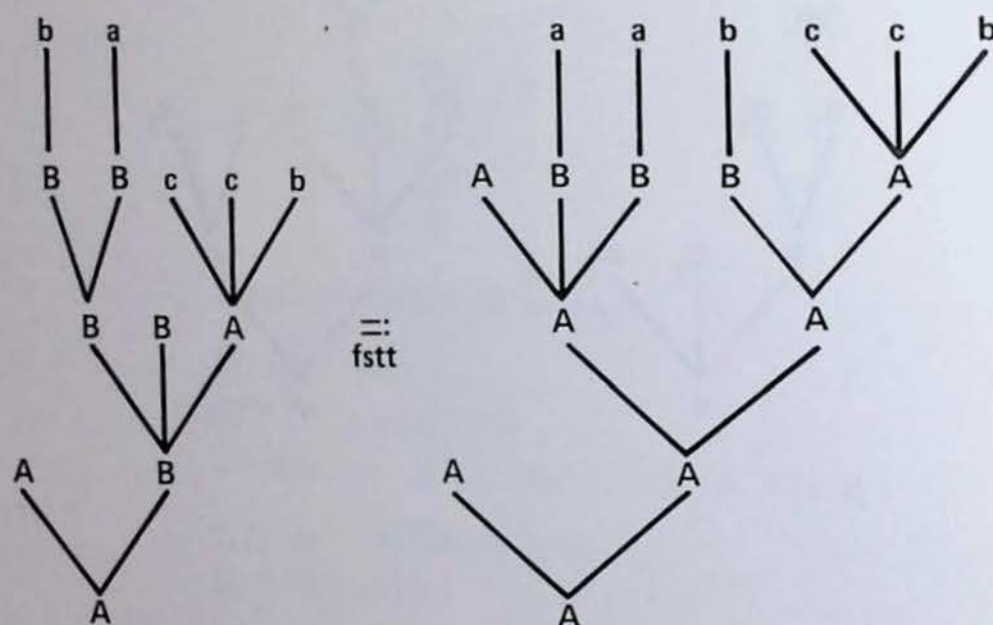


FIG. 3.2

tree transformation:



1. INTRODUCTION

1.1 Motivation, Problem and Justification

In the next three sub-sections we shall briefly attempt to motivate the present study, to very roughly outline the problem at hand and to justify our attacking this area.

1.1.1 Motivation

The semantics of several widely distinct programming languages have been given a rigorous intensional -- or operational -- definition which can be viewed as a class of process descriptions based on tree structures and involving transformations on these. These languages include LISP 1.2 (McCarthy [Ref. 12]), AE/ISWIM (Landin [Refs. 6-7]), RED/2 (Backus [Ref. 2]) and PL/1 (Lucas et.al. [Ref. 9]).

1.1.2 Problem

We therefore decided to formulate a system for defining such tree structures and for the specification of tree transformations. Then, given such a system, we investigate its properties and applicability.

1.1.3 Justification

We seek a unification of methods for the specification of the operational interpreter-oriented semantics of classes of programming languages. We are capable of relating such unified methods to the concept of tree automata [Refs. 1,14, 16-18] and finite state tree transformations [Refs. 14,18]. We generalize these notions and show that the generalization is applicable to the above semantic descriptions.

Our tree transformation systems amount to a class of programming languages, with programs being a finite set of possibly ordered tree transformation rules and with the input to such a program being a tree of some well-defined type. We can, using a tree transformation system, then define various evaluators for such languages.

The twin reason for our presenting (several) such evaluators is first to demonstrate the general and practical usefulness of the fstt and FSTT concepts, second to relate to current work on the properties of various computation rules presently being investigated due to their relation to the problem of fixed-point computations [Ref. 10]; these rules are in the context of e.g. ALGOL usually called: call by value, call by name etcetera; but generally we can label them: (leftmost) innermost-, (leftmost) outermost-, standard-, normal- and Kleene- computation rules [Refs. 10, 15].

We then give an algorithm which when applied to a tree

transformation system, i.e. a program in some tree programming language, will yield a "realization" in terms of a finite state 'tree walking' machine of a simple kind. Given a serial deterministic evaluator for the tree programming language one can further let a very simple modification of the constructed machine carry out all transformations on the tree; that is: a finite state machine suffices due to the tree on which it operates containing all the 'stacking' information (otherwise) required (to be stacked by a push-down machine).

The trees used throughout have all their nodes labelled. The labels belong to two disjoint alphabets: the alphabet of constant (or: syntactic) E symbols, and of operator (or: semantic) F symbols. A tree computation is a set of transformations turning a constant tree into a constant tree, with intermediate transformations taking place on trees containing both E and F symbols. For certain tree programming systems we can then show how to convert the modified finite state tree walking machine into a simple push-down tree walking machine which operate throughout on constant trees effecting the same final trees as would the above modified machine.

The 'constancy' of both E and F symbols (denoting just themselves) is in the tradition of Curry's Combinatory Logic [Ref. 4], with the f-operator symbols corresponding somewhat to the combinators in Curry's system. Thus one can view these programming languages as variable free systems and as suitable carriers for the definition of complete languages, whether primary or secondary. Such language

classifications has recently been investigated by Backus [Ref. 2]. The present report approaches this subject from (the restricted, albeit equally powerfull) viewpoint of the formulas of these languages all being tree structured, thus all being 'parenthetic'. It is the hope that we hereby are able to provide an additional framework for the theoretical investigations of such languages; to contribute some of these ourselves and to indicate means of mechanically implementing efficient (machine-oriented) interpreters for such -- and other -- languages; including those of 'applicative' nature [Refs. 6, 15]. In this report we shall not cover the so-called 'imperative' aspects of programming language (linguistics) [Ref. 7].

1.2 A Theory of process description

This paper is also an attempt to provide a 'theory of process description' for processes and computational structures in a form acceptable to processors [Ref. 20]. Central to such a theory is the structure of the sets of objects of the theory, the structure of the individual objects themselves, the set of auxiliary and primary operators defined on such structures and the explication on when and where to apply these operators.

The thesis of the paper is that the intuitive understanding and the formal development of the operational aspects of the semantics of certain classes of programming languages can be furthered by the representation of programs and data -- and of the computations

they invoke -- by means of trees and transformations on these.

1.3 Relation to other work

The obvious first relation is to the class of variable free programming languages recently reported by Backus [Ref. 2]. Going further back in time, into the realms of automata theory and linguistics we find the following relations:

Tree transformation systems of the kind that we are defining and exploiting can on the one hand be seen as a generalization first of the syntax directed transductions (Lewis and Stearns [Ref. 8] and Aho & Ullman [Ref. 1]), secondly of the finite state tree transformations (Rounds [Ref. 13] and Thatcher [Refs. 15-16]); on the other hand they can be viewed as models of transformational grammars (Chomsky [Ref. 3]). Our work is a direct generalization of Rounds and Thatcher only in the generality of our finite state tree transformation rules, which they call transition rules (of an automaton). Through this generality we are able to model both the upward and the downward movement of tree automata on trees and thus we are capable of achieving general computability. The tree automata resulting from our realizations are somewhat different from those of Aho and Ullman [Ref. 1].

1.4 Linguistic- and

Programming Language Semantics

Linguistically, one considers the 'full' grammar of the language to consist of two parts: a 'base grammar', which is typically context-free, and a 'transformational component', which operates on the output of the base component, i.e. in the context-free case the transformations are defined on derivation trees. Here there is some divergence between the formulation of (natural language) linguists and that of computer scientists. Linguistically, the semantics of the (natural language) sentence are fully specified by the derivation trees in the base component, the transformational component providing the actual sentences through "semantically vacuous", though linguistically significant changes. In contrast, the base component in tree automata studies has been the context-free programming languages, or rather their (oftentimes highly syntactically 'sugared') grammar (most of which is usually considered "semantically vacuous"), while the semantics are specified by the transformational component which either produces 'machine code', or 'directly', through interpretation, effects the meaning.

1.5 Rigorous Semantics

This section is of basic interest to the goals of our using tree transformations and tree automata for the rigorous, intensional, semantics of programming languages. It can, however, be skipped entirely, at least in a first reading.

Without defining our concepts rigorously at this stage we will here discuss some current methods for the precise description of the semantics of various programming languages. The reason for this early discussion is to relate these semantic 'formalizations' to the concept of tree transformations and to the workings of tree automata.

The languages -- or the semantic systems -- covered will be, in order: pure LISP 1.5 [Ref. 12], Landin's Applicative Expressions, AE [Ref. 6], LULER [Ref. 17] and Knuth's [Ref. 5] method. It would be going too far also to take up the aspects of the PL/1 formal semantic description [Ref. 9] which (certainly in many parts) relate to our subject.

The semantic definition of LISP is given as a LISP program itself. The language allows for the value of S-expressions to be functions, themselves applicable to other S-expressions. The semantics is therefore given as an interpreter providing for extensive execution time substitutions from (and to) the a-list environment. We shall give an FSTT rendition of the LISP semantics, which very closely follows the LISP evalquote function itself. There will however be some differences. These have to do first with the fact that our system essentially is a variable free programming language, then with the fact that we therefore must model, or state, explicitly the call by value feature assumed in LISP. It is in this connection that one should also see our precise specifications, as FSTE's, of various evaluation schemes: evaluate -- or substitute --

left-most inner-most, left-most outer-most, ... etc., -expressions first. A program in LISP is a 'doublet' of (two) S-expressions: a function and a list of arguments. In the course of evaluating this program the interpreter (quickly) transforms that pair into a LISP non-program 'triple' of S-expressions: a function part, an argument part and the a-list environment. Finally the interpreter returns another LISP non-program, now a 'single' S-expression as the result of the evaluation. It is our desire to model exactly such transformations, as trees: with first 2 subtrees, then 3, and finally just 1 subtree emanating from the root.

For precisely the same reasons as LISP, Landin's AE's are basically interpretively executed. Again call is by value, and inner-most replacement of AE's with their value is thus assumed. An AE program is an AE itself. The semantics for AE's is given twice (by Landin; see F. Lockwood Morris [Ref. 13] and Reynolds [Ref. 15] for treatments of hierarchies of interpreter to compiler semantics). First as an (interpreter) AE which when applied to a (program) AE 'winds' its way into the (program) AE such that, at any time in the evaluation process, the 'internal' representation is always an (instantaneous) AE. (Thus in Backus' words: AE's form a complete language). This is unlike LISP. The second definition, the SECD definition, of the AE semantics is 'mechanical', or intensional (or operative). It -- like LISP -- transforms an (the program) AE into a non-AE, a 'quadruple' of structures, none of which are AE's. We shall give an FSTT modelling the SECD semantics of AE's. Here the original AE will be taken to be (the yield of) a tree, and all

transformations are of trees into trees. (The result would then be the yield of the final tree, if any).

The EULER language was conceived as a generalization of ALGOL 60, which in turn is a typical non-interpretive language (in that highly compiler oriented program transformations can always be done). There are only a few run-time requirements: the checking of array bounds and the allocation and freeing of (data) storage upon entry to and exit from blocks. EULER however does have a number of features that call for extensive execution time inquiries on the part of the semantic system. The semantics is given in two steps. The first step basically amounts to the specification of 'compilations' of EULER programs into non-EULER programs. The second step basically amounts to the specification of execution time 'interpretation' of outputs from the first step. In the first phase, an EULER program, Q , is parsed, left-to-right, bottom-up -- and, in step with the recovery of the reverse right canonical derivation sequence of productions, corresponding sets of actions 'compile' Q into a data structure P . This part can then be modelled as a set of frontier-to-root finite state tree automata. With each meta-linguistic variable of the underlying grammar we need associate only 'synthesized' attributes. The sets of actions are expressed as EULER expressions; the set of 120 such sets is not, but could be, given as an EULER program. The resulting data structure P is of course a valid EULER data structure, but it is never an EULER program. Thus a tree, the phrase-marker for the input EULER program, is transformed into what basically is not a

tree, but a list of 'euler'-like tokens. These are then being scanned by the second part of the semantics. This part can basically be modelled as a finite state automaton (or sequential machine). The machine only scans P, never changes it; but it does scan it in both directions in order to effect the non-linear flow of control usually part of any (interesting) program. This machine -- i.e. the second part of the semantics -- then directs that transformations be made to a tree structured 'memory', S. Thus it is that we see the 'hidden' introduction of 'inherited' attributes; however in a rather non-transparent manner.

We do not pretend that we are going to model the semantics of EULER in this paper; but we do bring up the analogies here, not just to put our own work in perspective, but to indicate possible future directions for the Wirth/Weber and Knuth semantics. We now look at the latter:

Knuth generalize the semantics of Wirth & Weber. First his method assumes the availability of the complete, and unique, phrase-marker tree for any program Q. The manner in which it was derived, whether by bottom-up or by top-down analysis, is totally irrelevant to the semantics. Then to each syntactic class Knuth associates a finite set of 'synthesized'- and a finite set of 'inherited' attributes. The semantics can then basically be modelled as a set of finite state tree automata walking up and down the tree, in a manner which is synchronized by the automata themselves. No changes take place to the tree itself, but the various automata (or better: machines)

do initialize and update variables at each node and associated with the attributes declared for this node. The result of an evaluation is indicated by the final update -- and is then the value of -- one particular variable associated with the 'goal' symbol and located at the root of the tree.

It has been the purpose of this lengthy section to prepare for the argument that many seemingly diverse methods for the semantic specification of languages can be modelled as transformations of trees effected by the traversal of these trees by sets of finite state automata. In the case of LISP these automata would distribute themselves along paths of the trees, with only the one farthest away from the root being active (call by value, inner-most), while all others were kept suspended temporarily. The path of active automata would, however, be one that, dynamically speaking, corresponded to the movement of one or more automata in both directions on the tree. And with some circumscription the same can, and will later, be said about the other systems described above.

The available literature on tree transformations and tree automata have not attempted to model the both-directions movability of the automata. This paper is an attempt to provide such capabilities and to investigate some of the problems involved.

1.6 Disclaimer

The research reported on here is in a certain sense incomplete. We

do indicate some theoretical properties, but in this -- the first part -- we have deliberately put off such theoretical and analytical investigations in favor of design oriented constructions, experiments and synthesis. In a way we desired to go far enough down the road to demonstrating practical applicability of our concepts before embarking on more scholastic endeavors. The constructions all have been carried out with the aim of making them as general as possible; we have avoided we believe most if not all 'ad hoc' exceptions, thereby arriving at as few distinct classes of objects and as few operations on these as we believed reasonable. In doing so we hope not only to allow for the inclusion of further capabilities of our systems, but also to enable a meaningful theory to emerge from future investigations.

1.7 Organization of Part I

Section 2. contains most of the (rather trivial) definitions used throughout the report. Section 3. the definitions of 'finite state transitions' (rules), 'direct generations', 'finite state tree transformation systems, FSTTs' and of the 'tree computations' they invoke. Section 4. introduces the 'finite state tree evaluators, FSTEs' and give three extensive examples, and thus also examples of FSTTs. Section 5. takes a closer look at these evaluators for the sake of characterizing the 'evaluator-evaluator' processing these FSTEs. Thus 'non-deterministic-', 'deterministic-', 'partial-', 'total-' and 'reduced' FSTTs are defined -- and likewise the 'determinate-', 'serial-' and 'free-' computations invoked by

FSTTs. Section 6. relates our FSTTs to the theory of Tree Automata and Tree Transformations. Section 7. introduces the use of FSTTs as programs in the 'tree programming language' being defined, and also makes a concession to past practice in defining 'context-free tree programming systems'. Finally, in section 8., we introduce our notion of a 'serial finite state tree walking machine' and we give an algorithm for constructing such minimal state machines given an ordered set, $\underline{C}$, of fstt's.

In the appendix we give three examples of FSTTs: for the semantics of pure LISP 1.5, AE's and RED/2. For both RED/2 and an FSTE we then graphically illustrate our finite state tree transitions (rules) as a set of pairs of trees. In fact, this is the manner in which we 'program'; from the trees we then read off the linear description of these. Finally we give the machines realizing the RED/2 FSTT.

(The plan for part II of this report is the following: Section 9. brings the algorithm for converting the above finite state tree walking (serial) machine, first to a modified finite state tree machine, then to a push-down tree machine operating on trees with no $\underline{F}$ labels. Section 10. relates the tree programming systems to the class of primary- and secondary- languages [Ref. 2], and section 11. finally attacks the more theoretical issues raised by this work.)

2. DEFINITIONS

In this section we shall define certain classes of trees and a rather extensive set of operators defined on these, and on the results of having applied such operators to trees. The operators will return either (sets of) trees, (sets of) linear strings or symbols, or just some integer valued number.

Nodes of the trees will all be labelled. The labels belongs to either of three classes of alphabets. The alphabet, E, of 'constants' or 'syntactic classes'; the alphabet, F, of 'function names' or 'operator symbols' and the alphabet, Z, of 'variables'.

We will basically come to operate with two types of trees: complete trees, with nodes labelled from E or F, and variable trees, some of whose leaves are labelled with variables from (the finite set) Z. Complete trees will come to serve as our only data structure; the input-, the output- and the intermediate trees upon which our transformations take place. At each instance of a tree computation, the state of the computation, its 'instantaneous description' so-to-speak, will be completely defined by such a tree. Therefore one may view the state as the particular configuration of all the function names f over the entire tree. Indeed, as we shall see later, in section 6, such function symbols will then be referred to as state symbols. Finite sets of pairs of variable trees serve as

the set of instructions [Ref. 9] or the program operating on these instantaneous descriptions.

2.1 Alphabets

Let $\underline{U}$ be a finite set of symbols and $\underline{r}$ in $\underline{U}^* \underline{I}$ a ranking relation, then $[\underline{U}, \underline{r}]$ is a ranked alphabet. If $\underline{r}(u, n)$ holds for $u \in \underline{U}$ and $n \in \underline{I}$, then u has rank n . $\underline{U}_n$ denotes the finite subset of symbols of rank n . $\underline{E}$ and $\underline{F}$ denote two finite disjoint set of symbols. We shall usually (but not necessarily) take $\underline{U} = \underline{E} + \underline{F}$, $\underline{F}_0 = \emptyset$, $\underline{E}_0$ non-empty. $\underline{Y}$ is a finite set of 'arbitrary' variables, with $y \in \underline{Y}$ denoting any $u \in \underline{U}$. $[u, i] \in \underline{U}^* \underline{I}$ is a 'particular' variable, with $[u, i]$ denoting a distinct occurrence of a particular $u \in \underline{U}$, see [def. 2.3.9] and [remarks 3.3 and 6.1:6]. $\underline{Y} \text{ intersection } (\underline{U} + \underline{U}^* \underline{I}) = \emptyset$.

Remark 2.1

In the following we shall be using the meta-symbols: "(", ")", ",", ":", "[", "]" - these are not part of $\underline{E} + \underline{F} + \underline{Z}$.

2.2 Trees

The different classes of trees will be indicated by modifiers, these usually denote various sets - or an element of a set.

Definition 2.2.1 $\underline{T}(\underline{U}, \underline{Z})$

The set of U-expressions (-trees) with variables Z is defined inductively:

"if" $d \in \underline{U}_0 + \underline{Z}$

"then" $d \in \underline{T}(\underline{U}, \underline{Z})$

"if" (0) $n \in \underline{I}$

(1) $u \in \underline{U}_n$

(2) $\underline{t}_1, \dots, \underline{t}_n \in \underline{T}(\underline{U}, \underline{Z})$

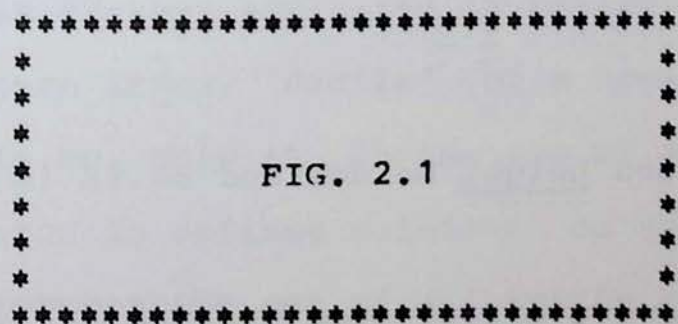
"then" $u(\underline{t}_1, \dots, \underline{t}_n) \in \underline{T}(\underline{U}, \underline{Z})$

where (normally) :

$$\underline{Z} = \underline{U} * \underline{I} + \underline{Y}$$

It follows that for $u \in \underline{U}_0$, $u() \equiv u \in \underline{T}(\underline{U}, \underline{Z})$.

Example FIG. 2.1



Definition 2.2.2 $\underline{T}(\underline{E}), \underline{T}(\underline{U})$

If $\underline{Z} = \emptyset$, the empty set, then $\underline{T}(\underline{U}, \underline{Z})$ by definition degenerates to $\underline{T}(\underline{U})$, the set of complete $\underline{U}$ -trees. $\underline{T}(\underline{E})$ is the set of constant $\underline{E}$ -trees.

Definition 2.2.3 $\underline{T}(\underline{U}, \underline{Z}, \underline{R})$

Given $r \in \underline{R}$, the set of trees in $\underline{T}(\underline{U}, \underline{Z})$ with root r is:

$$\begin{aligned}\underline{T}(\underline{U}, \underline{Z}, r) &\text{ def= } \{ t \text{ such that } t \in \underline{T}(\underline{U}, \underline{Z}) \ \& \ \text{top}(t) = r \} \\ \underline{T}(\underline{U}, \underline{Z}, \underline{R}) &= \text{set-union}(r \in \underline{R}) \ \underline{T}(\underline{U}, \underline{Z}, r)\end{aligned}$$

where top is explained in section 2.3. $\underline{R} \text{ in= } \underline{U} + \underline{Z}$.

Definition 2.2.4 $f\underline{T}(\underline{X})$

Where ' $\underline{X}$ ' is either ' $\underline{U}$ ' or ' $\underline{U}, \underline{Z}$ ' or ' $\underline{U}, \underline{Z}, \underline{R}$ '. Let $f \in \underline{F}$, then:

$$f\underline{T}(\underline{X}) = \{ t \text{ such that } t \in \underline{T}(\underline{X}) \ \& \ \text{top}[t] \in \underline{F} \}$$

Definition 2.2.5 $\underline{T}(\underline{U}; h)$

Let $h \in \underline{I}$, and height be defined as in [def. 2.3.13].

$$\underline{T}(\underline{U}; h) = \{ t \in \underline{T}(\underline{U}) \text{ such that } \text{height}[t] \text{ 'less than ='} h \}$$

That is, $\underline{T}(\underline{U}; h)$ is the set of all trees no higher than h .

2.3 Simple Operations on Trees, etc.

The number of operations to be defined on trees and the results from having applied some of these operators to trees is rather staggering. They are, however, all very simple, and only few of them are needed in section 3, a few more in section 4, and so forth. We have presented them all here, but encourage the reader to read them lightly and when more precision is needed to return for final consumption.

There are four classes of operators: the first returns subsets of alphabets, the second trees, the third strings and the last numbers.

'Top' of t returns the root label of a tree, 'nodes' return the set of all distinct labels of nodes in the tree, 'bud' return the set of labels of all leaves of a tree, 'var' return the set of labels of all variables of a tree, 'leaf' the corresponding set of all E leaves and 'yield' is further explained in remark 2.3.6. The next set of operators return trees. 'Prefix' of a tree is nearly what you would expect it to be, only it is the set of all such prefixes ! The 'suffix' operator is defined relative to various modifiers. The operator either returns the set of all suffix trees relative to some given prefix tree, pt , or it return the set of all suffix trees, independent of pt , i.e. taken over all pt ! An 'interior' tree is basically the prefix of some suffix tree. 'Projecting' the

nodes of a variable tree yields a complete tree (or a set of such trees), changing 'variable' labelled leaves into labels from $\underline{E}$ or $\underline{F}$. A 'path' of a tree is again nearly what you would expect it to be, only this time it again returns the set of all such complete paths. The 'length' of a string is the number of symbols it contains. The 'height' of a tree is the length of the longest path in that tree. With this informal description you may go on directly to section 3 !

Examples of the application of some of these operators are shown in examples, FIGs 2.1-2.

Definition 2.3.1 $\underline{\text{top}}[t]$

"if" $t \in \underline{E} + \underline{Z} + \underline{F}$

"then" $\underline{\text{top}}[t] = t$

"if" $t = u(\underline{t}_1, \dots, \underline{t}_n)$

"then" $\underline{\text{top}}[t] = u$

Definition 2.3.2 $\underline{\text{nodes}}[t]$

"if" $t \in \underline{E} + \underline{F} + \underline{U} * \underline{I} + \underline{Y}$

"then" $\underline{\text{nodes}}[t] = \{t\}$

"if" $t = u(\underline{t}_1, \dots, \underline{t}_n)$

"then" $\underline{\text{nodes}}[t] = \{u\} + \underline{\text{set-union}}(i) \underline{\text{nodes}}[\underline{t}_i]$

Definition 2.3.3 $\text{bud}[t]$

"if" $t \in \underline{L} + \underline{Z}$

"then" $\text{bud}[t] = t$

"if" $t = u(\underline{t}_1, \dots, \underline{t}_n)$

"then" $\text{bud}[t] = \text{set-union}(i) \text{ bud}[\underline{t}_i]$

Definition 2.4.3 $\text{var}[t]$

$\text{var}[t] \text{ def= } \text{bud}[t] \text{ intersection } \underline{Z}$

Definition 2.3.5 $\text{leaf}[t]$

$\text{leaf}[t] \text{ def= } \text{bud}[t] \text{ intersection } \underline{E}$

Definition 2.3.6 $\text{yield}[\underline{W}][t]$

"if" $t = u(\underline{t}_1, \dots, \underline{t}_n)$

"then" $\text{yield}[\underline{W}][t] = \text{yield}[\underline{W}][\underline{t}_1] * \dots * \text{yield}[\underline{W}][\underline{t}_n]$

"if" $t \in \underline{W}$

"then" $\text{yield}[\underline{W}][t] = t$

"else" $\text{yield}[\underline{W}][t] = \text{null}$

Remark 2.3.6

$\text{yield}[\underline{E0}][t]$ is the string (the left-to-right concatenation) of symbols of rank 0 in the order in which they appear in t . $\underline{W}$ is an alphabet, usually a subset of $\underline{E0} + \underline{Z}$. If $\underline{W} = \underline{E} + \underline{F} + \underline{Z}$ then we drop $[\underline{W}]$ from qualifying yield .

Definition 2.3.7 $\text{prefix}[t]$

"if" $t = \text{null}$ "then" $\text{prefix}[t] = \emptyset$

"if" $t \in \underline{E} + \underline{Z}$

"then" $\text{prefix}[t] = t$

"if" (0) $t = u(\underline{t}_1, \dots, \underline{t}_n)$

(1) $\underline{pt}_i \in \text{prefix}[\underline{t}_i]$

"then" $t, u, u(\underline{pt}_1, \dots, \underline{pt}_n) \in \text{prefix}[t]$

Remark 2.3.7

If $t \in T(\underline{U}, \underline{Z})$ and $\underline{pt} \in \text{prefix}[t]$, then $\underline{pt} \in T(\underline{U}, \underline{U} + \underline{Z})$.

Let T denote a set of trees, then:

$\text{prefix}[T] = \text{set-union}(t \in T) \text{ prefix}[t]$

Definition 2.3.8 $\text{suffix}[t, \underline{pt}]$

"if" $p_{\underline{t}} = t$ "then" $\underline{\text{suffix}}[t, p_{\underline{t}}] = \emptyset$

"if" (0) $t = u(\underline{t}_1, \dots, \underline{t}_n)$

(1) $p_{\underline{t}} = u$

"then" $\underline{\text{suffix}}[t, p_{\underline{t}}] = u(\underline{t}_1, \dots, \underline{t}_n)$

"if" (0) $t = u(\underline{t}_1, \dots, \underline{t}_n)$

(1) $p_{\underline{t}} = u(p_{\underline{t}_1}, \dots, p_{\underline{t}_n})$

(2) $p_{\underline{t}_i} \in \underline{\text{prefix}}[\underline{t}_i]$

"then" $\underline{\text{suffix}}[t, p_{\underline{t}}] = \underline{\text{set-union}}(i)$
 $\underline{\text{suffix}}[\underline{t}_i, p_{\underline{t}_i}]$

Auxiliary suffix operators:

$\underline{\text{suffix}}[t] = \underline{\text{set-union}}(\text{all } p_{\underline{t}} \in \underline{\text{prefix}}[t])$
 $\underline{\text{suffix}}[t, p_{\underline{t}}]/t$

$\underline{\text{suffix}}[T, p_{\underline{t}}] = \underline{\text{set-union}}(\text{all } t \in T \ \& \ p_{\underline{t}} \in \underline{\text{prefix}}[t])$
 $\underline{\text{suffix}}[t, p_{\underline{t}}]$

$\underline{\text{suffix}}[T] = \underline{\text{set-union}}(\text{all } t \in T) \underline{\text{suffix}}(t)$

Definition 2.3.9 interior(t)

interior[t] = $\{ \underline{it} \text{ such that:}$

$\underline{it} \in \underline{\text{prefix}}[t] + \underline{\text{prefix}}(\underline{\text{suffix}}(t)) \}$

If $t \in T(\underline{U}, \underline{Z})$ then an interior of t : $it \in T(\underline{U}, \underline{U} + \underline{Z})$.

Definition 2.3.10 node-projection $[t]$

Let $t \in T(\underline{U}, \underline{Z})$, where $\underline{U} = \underline{E} + \underline{F}$, $\underline{Z} = \underline{Y} + \underline{U} * \underline{I}$, then:

"if" $t = u$, $u \in \underline{U}$, "then" $\underline{n-p}[t] = t$

"if" $t = z = [e, i]$, $t \in \underline{U} * \underline{I}$

"then" $\underline{n-p}[t] = e$

"if" $t \in \underline{Y}$

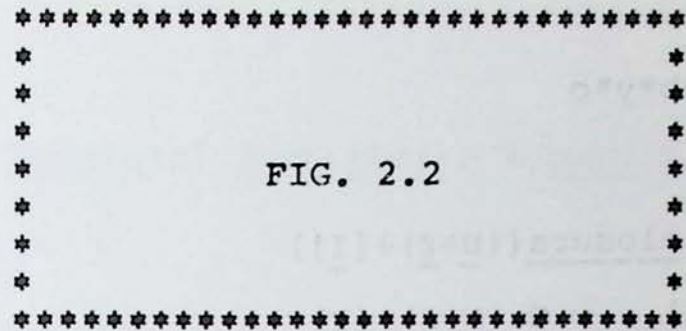
"then" $\underline{n-p}[t] = \underline{U}$

"if" $t = u(\underline{t}_1, \dots, \underline{t}_n)$

"then" $\underline{n-p}[t] = u(\underline{n-p}[\underline{t}_1], \dots, \underline{n-p}[\underline{t}_n])$,

Where we use $\underline{n-p}$ as short for node-projection. It is seen that the set $\underline{n-p}[t]$ in $T(\underline{E} + \underline{F}, \underline{E} + \underline{F})$.

Example FIG. 2.2



Notice that the node-projection of this tree is an interior tree of the tree shown in FIG. 2.1.

Definition 2.3.11 path[t]

"if" $t = w$, "then" path[t] = $w[0][0]$

"if" $t = u(\underline{t}_1, \dots, \underline{t}_n)$

"then" path[\underline{t}_i] * $u[i] \in \text{path}[t]$ all i

Where the concatenation is between a set of paths, an element $u \in \underline{U}$ and a branch selector [i] in I , resulting in a set of concatenated strings of symbols. Defined this way, a path $p \in \text{path}[t]$ is overly (redundantly) defined: by deleting all symbols $u \in \underline{E+F}$ in p , getting p' , we can still uniquely construct the corresponding path p once t is presented.

Definition 2.3.12 Disjoint and Contained trees

Let $t = w(\underline{t}_1, \dots, \underline{t}_n)$, then we say that the trees $\underline{t}_1, \dots, \underline{t}_n$ are disjoint. Let $p \in \text{path}[t]$, specifically:

$$p = a*x*b*y*c$$

where:

$$a, b, c \in \text{closure}((\underline{U} + \underline{Z}) + [\underline{I}])$$

$$x \in \underline{U}^*[\underline{I}], \quad y \in (\underline{U} + \underline{Z})^*[\underline{I}]$$

then x (y) define a tree, $\underline{t}_x$ ($\underline{t}_y$), for which:

$$x*b*y*c \in \text{path}[\underline{t}_x] \quad (y*c \in \text{path}[\underline{t}_y])$$

We say that the tree $\underline{t}_x$ contains the tree $\underline{t}_y$, $\underline{t}_y$ is contained by $\underline{t}_x$. Henceforth we shall mainly use containment for trees, including interiors, whose roots are in $\underline{F}$.

Definition 2.3.13 length[w]

Let w be a string of symbols in any alphabet $\underline{W}$:

$$\text{"if" } w = \underline{\text{null}}, \text{ "then" } \underline{\text{length}}[w] = 0$$

$$\text{"if" } w \in \underline{W}, \text{ "then" } \underline{\text{length}}[w] = 1$$

$$\text{"if" } w = v_1*v_2, v_1, v_2 \in \text{closure}(\underline{W})$$

$$\text{"then" } \underline{\text{length}}[w] = \underline{\text{length}}[v_1] + \underline{\text{length}}[v_2]$$

Definition 2.4.13 height[t]

Let $t \in \underline{T}(\underline{U}, \underline{Z})$, $\underline{W} = \underline{U}^*[\underline{I}]$, then:

height[t] = max(length[p] such that p $\in$ path[t])

3. TREE TRANSFORMATIONS

Having now defined complete and variable trees we can then go on to define the instructions, the fstt's, which make up a program, an FSTT. Both individual fstt's and the entire FSTT are meant to apply to complete trees and to yield, result in, complete trees.

Each instruction has an antecedent, lt, and a consequent, rt. If, in some informal sense (which will presently be made formal) some lt 'matches' some interior of a complete tree, then this tree may or will be transformed according to the specification given by the rt.

Definition 3.1 Finite State Tree Transition, fstt

An fstt is a pair:

$$[\underline{lt} =: \underline{rt}]$$

where:

$$\underline{lt} \in \underline{fT}(\underline{U}, \underline{Z})$$

$$\underline{rt} \in \underline{T}(\underline{U}, \underline{Z})$$

where:

$$\underline{U} = \underline{E} + \underline{F}, \quad \underline{Z} = \underline{Y} + \underline{U} * \underline{I}$$

such that:

$$\underline{var}[\underline{rt}] \subseteq \underline{var}[\underline{lt}]$$

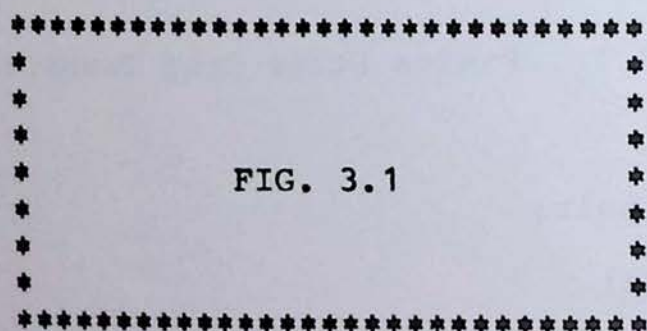
and such that in yield[lt] there are no two occurrences of any variable in var[lt].

Remark 3.1

The definition of fstt's is made such that the consequent provides for (1) the pruning-, (2) the permutation-, (3) the grafting of multiple copies- and (4) the introduction of entirely new subtrees. It is this combined set of capabilities which allows us to model recursive functions and thus to achieve general computability.

Example FIG. 3.1

An example of an fstt is shown in FIG. 3.1:



Observe that we have chosen to let lt if this fstt be the tree of FIG. 2.2.

Definition 3.2 Applicability

An fstt [lt =: rt] $\in \underline{M}$ is applicable (with respect to M) to a tree t" $\in \underline{T}(\underline{E}+\underline{F})$ if there exist a tree it in the interior of t"

such that:

$$\underline{it} \in \underline{n-p[lt]}$$

Definition 3.3 Assignment

An fstt containing variables will be made to stand for a whole, usually infinite, set of complete trees. Let $\underline{z}$ be a variable in an fstt, then by the definition it will appear exactly once in the $\underline{lt}$. If $\underline{z} \in \underline{Y}$ then we informally state that any complete tree can be 'substituted' in $\underline{z}$'s place. If $\underline{z} = [u, i] \equiv \underline{ui}$ form some distinct $u \in \underline{E+F}$, then any complete tree whose root is u can take the place of $\underline{z}$. If we perform such an 'assignment' to all variables in $\underline{lt}$ -- selecting arbitrarily one tree for each variable -- then we arrive at a complete tree, call it $\underline{s[lt]}$, where $\underline{s}$ will be defined. $\underline{s[lt]}$ is furthermore said to define a particular set of assignments of complete trees to variables in the fstt. More precisely:

Given (0) $\underline{t} \in \underline{T(E+F)}$

(1) fstt: $[\underline{lt} =: \underline{rt}]$

such that

(2) $\underline{t'} \in \underline{\text{suffix}[t]}$

(3) $\underline{it} \in \underline{\text{prefix}[t']}$

i.e. $\underline{it} \in \underline{T(U, U)}$

(4) $\underline{it} \in \underline{n-p[lt]}$

Then - for a given such $\underline{it} - [t, \text{fstt}, \underline{it}]$ defines an assignment of values in $\underline{T(E+F)}$ to the variables in $\underline{Z}$:

$$(5) \underline{a}[t, \text{fstt}, \text{it}]: \underline{z} := \underline{T}(\underline{E} + \underline{F})$$

where we usually omit the qualification $[t, \text{fstt}, \text{it}]$.

First observe that there is a unique mapping, g , (isomorphism) from lt to it . In the fstt , $[\text{lt} =: \text{rt}]$, let z be in $\text{var}[\text{lt}]$, then there may be zero, one or more (though a finite number of) z in $\text{yield}[\text{rt}]$. Accordingly, any variable, z , in lt selects (under g) a symbol in $\text{yield}[\text{it}]$ and thus a symbol, u , in t . This symbol is the root of a particular tree t'' in $\text{suffix}[t', \text{it}]$. In short:

$$(6) \underline{a}[t, \text{fstt}, \text{it}][z] = t'' \text{ , i.e. } \underline{a}[z] = t''$$

Remark 3.3

We use $[u, i]$, often written $\underline{ui}$, as a variable whenever we wish to fix the class ($\underline{E}$ or $\underline{F}$) and instance (of a particular e or f) of a symbol - therefore the word 'particular' variable; otherwise we use subscripted $\underline{y}_i \in \underline{Y}$, a y then denotes any symbol in $\underline{E} + \underline{F}$.

The latter notation is purely technical, it permits one fstt to stand for a whole class of fstt 's. One could further refine the notation if so desired for further convenience of 'programming' in fstt 's by 'splitting' $\underline{Y}$ into e.g. three sets ($\underline{Y}_e$, $\underline{Y}_f$ and $\underline{Y}_{ef}$) whose symbols would range over all e 's, all f 's and all e 's and f 's.

Definition 3.4 Substitution

This assignment in turn defines a substitution, $\underline{s}$, of trees in suffix $[t', \underline{t}]$ for variables, z , in $\underline{lt}$ and $\underline{rt}$. In particular, $\underline{s}$ maps t'' in t for one z in $\underline{lt}$, and zero, one or more (though a finite number of) z in $\underline{rt}$:

(7) "if" $t = u$, "then" $\underline{s}[t] = t$

"if" $t = z$

"then" $\underline{s}[t] = \underline{a}[t] = \underline{a}[z]$

"if" $t = u(\underline{t}_1, \dots, \underline{t}_n)$

"then" $\underline{s}[t] = u(\underline{s}[\underline{t}_1], \dots, \underline{s}[\underline{t}_n])$

Consequently:

(8) $\underline{s}(\underline{lt}) \in f_{\underline{T}(\underline{E}+\underline{F}, \underline{T}(\underline{E}+\underline{F}))}$

(9) $\underline{s}(\underline{rt}) \in \underline{T}(\underline{E}+\underline{F}, \underline{T}(\underline{E}+\underline{F}))$

Definition 3.5 Direct Generation

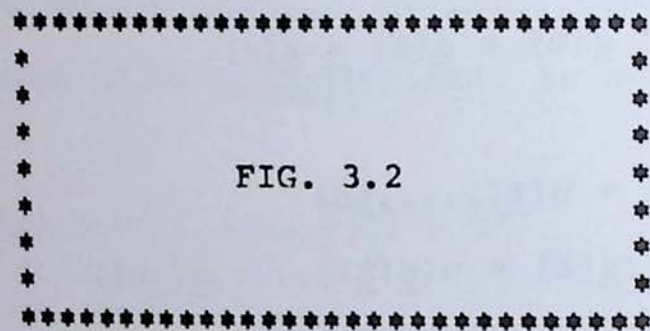
Given (0-6) of the previous definitions, t'' is directly generated, $==$, from t by replacing t' in t with $\underline{s}(\underline{rt})$:

$t == t''$ under $fstt$

The conditions (0-4,7) define a relation, $\equiv$, on $\underline{T}(\underline{E}+\underline{F})$. Let $\equiv^*$ be the reflexive, transitive closure of $\equiv$. Let $\equiv^+$ denote a generation sequence involving at least one direct generation.

Example FIG. 3.2

Fig. 3.2 illustrates the application of the fstt of FIG. 3.1 to the tree of FIG. 2.1:



Definition 3.6 Finite State Tree Transformation

An f.s.t. transformation (FSTT), $\underline{A}$, is a 4-tuple:

$$\underline{A} = [(\underline{E}+\underline{F}, \underline{r}), \underline{F}, \underline{f_0}, \underline{M}]$$

where $\underline{f_0} \in \underline{F}$, and $\underline{M}$ is a finite set of fstt's of the form defined above.

Definition 3.7 Proper FSTTs

For $\underline{A}$ to be a proper FSTT we shall require:

(1) that:

$$\text{set-union}(i) \text{ top}(\text{lti}) = \underline{F}$$

for all fstt's in $\underline{M}$. Thus $\underline{f_0}$ is guaranteed to be the top of some lti .

(2) that the set of all fstt's define a set, $\underline{N}$, of symbols in $\underline{E+F}$:

$$\underline{N} = \text{set-union}(i) \{ \text{nodes}[\text{lti}] + \text{nodes}[\text{rti}] \}$$

Those symbols of $\underline{N}$ not in $\underline{F}$ by this definition belongs to $\underline{E}$, thus:
 $\underline{N} \text{ in} \underline{E+F}$. Henceforth we shall assume all FSTTs to be proper.

Definition 3.8 Tree Computation

Let $t, t' \in \underline{T(E)}$ and $\underline{A} = [(\underline{E+F}, \underline{r}), \underline{F}, \underline{f_0}, \underline{M}]$, then t' is a computation of t if $\underline{A}$ allow a generation sequence:

$$\underline{f_0}(t) =+= t'$$

Definition 3.9 Tree Mapping

The tree mapping defined by $\underline{A}$ is the set of pairs:

$$\underline{D} = \{ (t, t') \in \underline{T(E)} * \underline{T(E)} \\ \text{such that } \underline{f_0}(t) =+= t' \}$$

Remark 3.

In the computations defined above [def. 3.8] there was no

commitment as to the order in which direct generations were performed. Given a tree $t \in \underline{T}(\underline{E}+\underline{F})$ two basic situations arise. The first has to do with the computation over contained trees, the other with computation over disjoint trees.

Let $t \in \underline{T}(\underline{E})$, then $\underline{f_0}(t) \in \underline{T}(\underline{E}+\underline{F})$, with exactly one f , namely $\underline{f_0}$, in $\underline{F}$ in $\underline{f_0}(t)$. After one or more direct generations we get a tree $t' \in \underline{T}(\underline{E}+\underline{F})$, with normally several $f \in \underline{F}$ in t' . In general there may be several symbols in $\underline{F}$ along any one path in $\text{path}[t']$. Each such $\underline{f_{ij}}$ define a subtree, $\underline{it}$, in the interior of t' ; therefore one may test any such $\underline{it}$ for applicability. At any one instance in a derivation sequence several such interiors may be applicable. This raises several questions. In which order should we apply the direct generations -- does it matter whether we impose a certain order of evaluation. As defined above, tree computations specify no order of preference. It is the specific purpose of the evaluators - the subject of our next two sections - to define exactly this order - whenever it may matter.

The first two evaluators specify that the direct generations be performed on maximally containing (i.e. outermost) interiors satisfying the applicability criteria - if any are found. The third evaluator specifies that the direct generation be performed to interior trees $\underline{f}(\underline{t_1}, \dots, \underline{t_n}) \in \underline{fT}(\underline{E})$, i.e. where all i , $\underline{t_i} \in \underline{T}(\underline{E})$.

Let $t = e(\underline{t_1}, \dots, \underline{t_n})$, all $\underline{t_i} \in \underline{T}(\underline{E}+\underline{F})$. Again [def. 3.8] does not specify the order in which the disjoint trees $\underline{t_i}$ should be subject

to direct generations. In the first evaluator [def. 4.3] we assume that the computations on $\underline{t}_i$ for all i are independent, i.e. lead to the same constant tree, $\underline{t}'_i$, regardless of the order of generations. In the subsequent two evaluators [defs. 4.4-5] we specify that the leftmost (non-constant) tree be computed first.

An FSTT applies to a set of trees $\underline{T}(\underline{E})$. To each operator symbol, $f \in \underline{\Gamma}$ there corresponds one or more fstt's in $\underline{M}$, namely those for which $\underline{\text{top}}$ of the left part returns that f . The subset of all fstt's having $\underline{\text{top}}[\underline{lt}] = f$ define a function which we shall also call f . Thus an FSTT can be seen as a 'program' whose main procedure is the function $\underline{f}_0$ with $\underline{f}_0$ being defined in terms of a finite number of 'inner' procedures. These are all 'declared' on the same level as $\underline{f}_0$, but singling out $\underline{f}_0$ as an initial function brings about the distinction between main and inner or 'auxiliary' functions or procedures. For $\underline{f}_0$ to be applicable to the root of some, but of course not any, constant tree in $\underline{T}(\underline{E})$ we must require that at least one $\underline{lt}_i$ with root $\underline{f}_0$ has no other occurrences of operator symbols f in $\underline{lt}_i$ than that one, the root, $\underline{f}_0$. The question therefore naturally arises whether there are functions in $\underline{M}$ which, regardless of the form of our initial constant input tree, will ever be invoked. We shall attempt to answer this and other questions as we continue with the next sections.

Alternative 3.10 [Def. 3.8]

Definition 3.8 is a restriction of a more general definition of

computation for which we allow the input tree t and the result tree t' to be just a complete tree, i.e. in all of $\underline{T(E+F)}$ rather than the subset $\underline{T(E)}$. If a t' has one or more nodes labelled with some f 's in $\underline{F}$, but no applicable interiors, then t' would, according to the generalized definition, be the result.

We have not chosen this alternative and more general definition; for several reasons some of which we will mention. First -- choosing or not choosing the alternative -- it would make little if any substantial change to our subsequent definitions and constructions. Then we find that such a generalization is not really warranted in order to satisfy our desire to model adequately and economically the interpreter based operational semantics definition of the languages earlier mentioned. We shall, however, in section 10, have occasion to take up again the subject of this generalization.

Discussion 3.11 On the presence of f -Operators

It can be argued that our definition of transformations is not as general as it could be. We could e.g. choose not to avail ourselves of a disjoint set of operators, but instead just have one set, $\underline{E}$. For one thing the definitions -- and indeed some of the subsequent constructions -- would be somewhat simplified. We do, however, believe that we would thereby have lost some of the capabilities the system, as it now stands, has in faithfully modelling actual programming practices.

4. TREE EVALUATIONS

This section presents an investigation -- from a tree transformation point of view -- of various orders of evaluation (or orders of application, or computation rules), usually in conventional programming languages referred to as 'call by value', 'call by name', etc., [Refs. 10, 15]. The essence of our results is that we can express any such evaluation rule as a particular FSTT, an FSTE. And the question then naturally arises as to the nature of the universal evaluator-evaluator which applies to such FSTE's. Some characterization will be given in this part of the report, with more to follow in part II. The importance of fixing and/or controlling the evaluation rules have recently been systematically investigated by Manna et. al. [Ref. 10].

Definition 4.1 Universal f-Inspection Schemes

A Universal f-Inspector, V, is a rule, s, of substitution - as defined in 3.4 - and an 'incomplete' FSTT, B, basically of the form defined in 3.6.

$$\underline{V} : [\underline{s}, \underline{B} = [[\underline{E}' + \underline{F}', \underline{r}'], \underline{F}', \underline{\text{meaning}}, \underline{1}']]$$

B provides a rule of inspection of f's in F, for the class, FSTT, of all FSTTs A.

Remark 4.1

The idea behind $\underline{V}$ is to sequence the order in which interiors of a tree $t \in \underline{T}(\underline{E}+\underline{F})$ are subjected to direct generations; these latter are not defined by $\underline{V}$, but by the particular FSTT ($\underline{A}$) which is used in 'completing' $\underline{V}$. The 'incompleteness' of $\underline{B}$ amounts to the lack of fstt's properly belonging in $\underline{A}$. $\underline{V}$ then, is to be viewed as a 'skeleton' upon which, or a 'shell' inside which, any one - of a wide class of - FSTT's, $\underline{A}$, may be fitted. In any actual evaluator [def. 4.2] $\underline{A}$ must be properly substituted into those fstt's in $\underline{B}$ that generally stand for $\underline{A}$. This somewhat imprecise definition should suffice for the time being. In section 5.8 we then give the precise construction for the substitution of (any) $\underline{A}$ into $\underline{B}$. In the following we shall define several such evaluators. As a concession to past practice, we shall however present $\underline{M}'$ in a syntactically 'sugared' form. In later sections we shall show the simple conversions involved in giving the fstt rules. $\underline{F}'$ will contain $\underline{F}$ as a proper subset.

Definition 4.2 f.s.t. Evaluator

An f.s.t. Evaluator (FSTE), $\underline{V}[\underline{A}]$ ($\underline{V}$ of $\underline{A}$), is a 'completion' of a universal f-inspector scheme $\underline{V}$ with respect to a particular FSTT $\underline{A}$. The completion process will subsequently be defined. The FSTE, $\underline{V}[\underline{A}]$, define a restricted class of computations, $\underline{V}[\underline{A}]$ -evaluations, on trees, $t \in \underline{T}(\underline{E}+\underline{F})$:

$$\text{meaning}[\underline{f0}[t]] =* t'$$

where:

$$t, t' \in \underline{T}(\underline{E})$$

meaning defined by V, f0 defined by A

We shall often write: $\underline{V}[\underline{A}[t]]$ as short for the generation sequence above leading from t to t' .

Notation 4.2

The form of the following few definitions is practically the same. Instead of tediously repeating very similar conditions these will be defined here by establishing equivalence between some $: \# :$ and a form of a tree in $\underline{T}(\underline{E}+\underline{F})$.

$$:0: \quad \text{if } t \in \underline{T}(\underline{E})$$

$$:1: \quad \text{if } t = f(\underline{y1}, \dots, \underline{yn})$$

$$:2: \quad \text{if } t = e(\underline{y1}, \dots, \underline{yn})$$

$$:a: \quad \underline{it} \in \text{prefix}[t] \quad \&$$

$$\underline{lt} =: \underline{rt} \in \underline{M} \quad \&$$

$$\underline{it} \in \underline{n-p}[\underline{lt}]$$

$$:b: \quad \text{all } i \text{ 'less than' } m \text{ 'less than' } = n$$

$$\underline{a}[\underline{yi}] \in \underline{T}(\underline{E}) \quad \& \quad \underline{a}[\underline{ym}] \notin \underline{T}(\underline{E}), \text{ i.e. } \underline{a}[\underline{ym}] \in \underline{T}(\underline{E}+\underline{F})$$

:c: all $i, a[yi] \in T(E)$

:3: if :1: & :a:

:4: if :1: & not :a:

:5: if :1: & :b:

:6: if :1: & :c: & not :a:

:7: if :2: & :b:

:8: if :1: & :c:

We shall furthermore be using the following symbols: $\underline{f0}, \underline{D}, \underline{X}, \underline{D1}, \underline{D2}, \dots$ etc., not to denote subscripted variables, but distinct symbols, mostly in $\underline{F}$ and $\underline{F'}/\underline{F}$.

Remark 4.2.1 Completion

The propositions :a:, :b: and :c: themselves require elaborately defined FSTT's. In [defs. 4.3-5] we will assume that all these can - and are - defined. In section 5. we shall then modify [defs. 4.4-5] to include fstt's carrying out the tests required by propositions :b: and :c:. In section 8 we shall then show how to construct a realization of the FSTT corresponding to :a:. Once we have provided the (sub)FSTT's for :a:, :b: and :c: and substituted them into the "meta-fstt's" used below (i.e. all those involving propositions :a:, :b: and :c:), then we have completed the proper construction of $\underline{V}[\underline{A}]$, and the $\underline{B}$ part is then no longer incomplete - the result is then, not an f-inspection scheme (for a class of $\underline{A}$'s), but an evaluator.

Remark 4.2.2

In the above [notation 4.2] and in most subsequent FSTT constructions, [defs. 4.3-5, 5.5-7] we follow the convention that the use of e , f and $\underline{y}_i$ denote any $e \in \underline{E}$, $f \in \underline{F}$ and any $[u,i] \in \underline{U}^* \underline{I}$, respectively. This applies also to the occasional use of $\underline{e}_i$ and $\underline{f}_j$, for which the e and f part ranges accordingly.

Example 4.3 Canonical Outermost Evaluator, $\underline{V}_{co}$
(first version)

The primary distributor, $\underline{D}_1[t]$ (= meaning) :

- (0) :0: =: t
- (1) :1: =: $\underline{x}_1[t]$
- (2) :2: =: $e(\underline{D}_1[\underline{y}_1], \dots, \underline{D}_1[\underline{y}_m])$

The primary transformer, $\underline{x}_1[t]$:

- (3) :3: =: $\underline{D}_1[\underline{s}[rt]]$
- ((4) :6: =: $\underline{?}$)
- (5) :4: =: $\underline{x}_1[f(\underline{D}_2[\underline{y}_1], \dots, \underline{D}_2[\underline{y}_m])]$

Remark 4.3.1

It will soon be clear that - in these formulas - action on the

innermost operators of $\underline{F'}$ precedes that of the enclosing operators of $\underline{F'}$. In this case all $\underline{D2}$ are carried through before $\underline{X1}$ is acted upon. This is consistent with any $f' \in \underline{F'}$ being defined (in (#)) on trees in $\underline{T(E+\underline{F}, \underline{Z})}$ only. We shall later, however, desire not to exclude $\underline{F'}$, in fact we shall subsequently want to have $f' \in \underline{F'}$ defined on trees in $\underline{T(E+\underline{F'}, \underline{Z'})}$, where $\underline{F'}$ includes $\underline{F}$, and with $\underline{Z}$ suitably extended to $\underline{Z'}$!

The secondary distributor, $\underline{D2}[t]$:

$$(6) \quad :0: \quad =: \quad t$$

$$(7) \quad :1: \quad =: \quad \underline{X2}[t]$$

$$(8) \quad :2: \quad =: \quad e(\underline{D2}[\underline{y1}], \dots, \underline{D2}[\underline{yn}])$$

The secondary transformator, $\underline{X2}[t]$:

$$(9) \quad :3: \quad =: \quad \underline{s}[\underline{rt}]$$

$$((10) \quad :6: \quad =: \quad ?)$$

$$(11) \quad :4: \quad =: \quad f(\underline{D2}[\underline{y1}], \dots, \underline{D2}[\underline{yn}])$$

Remark 4.3

[def. 4.3] defines what is meant by 'outermost' evaluation. If $\underline{X1}$ as invoked by (1) fails to transform, then (5) allow 'sub-outermost' evaluation under $\underline{D2}$. $\underline{D2}$ leads to $\underline{X2}$, (7). When $\underline{X2}$ succeeds, (9), it will not result in a renewed evaluation of t ,

note the absence of any $f' \in \underline{F}$ in (9), whereas the corresponding (3) specifies $\underline{D1}$! Instead computations will 'bounce' back to the outer $\underline{X1}$ as specified by (5). This 'bouncing' necessitates two levels of distributors and transformers, and is found in essentially the same form in [def. 4.4].

In order for (5) to complete its job it is assumed that the application of $\underline{D2}$'s will change t such that $\underline{X1}[t]$ falls through under (3). If not, then (4) and (10) will, if included in $\underline{M}$, eliminate such blocked trees (see [Def. 5.4]). The parenthesization of these four fstt's in $\underline{M}$ indicate that they are optional. If included in $\underline{M}$ they will - if ever invoked - make up for an otherwise 'partially specified' $\underline{M}$ (see [def. 5.3]). By including these rules we have effectively extended $\underline{E}$ to include the undefined symbol, $\underline{?}$. Rather than letting $\underline{M}$ insert the $\underline{?}$ one may, when defining $\underline{A}$ choose to complete $\underline{M}$ by making it a total FSTT (see [Def. 5.3]).

Example 4.4 Normal Evaluator, $\underline{Vn}$ (first version)

The primary distributor, $\underline{D1}[t]$ (= meaning) :

- (0) :0: =: t
- (1) :1: =: $\underline{X1}[t]$
- (2) :7: =: $\underline{D1}[e(\underline{y1}, \dots, \underline{ym-1}, \underline{D1}[\underline{ym}], \underline{ym+1}, \dots, \underline{yn})]$

The primary transformer, $\underline{X1}[t]$:

- (3) :3: =: $\underline{D1}[s[rt]]$
 ((4) :6: =: $\underline{?}$
 (5) :5: =: $\underline{X1}[f(\underline{y1}, \dots, \underline{ym-1}, \underline{D2}[\underline{ym}], \underline{ym+1}, \dots, \underline{yn})]$

The secondary distributor, $\underline{D2}[t]$:

- (6) :0: =: t
 (7) :1: =: $\underline{X2}[t]$
 (8) :7: =: $e(\underline{y1}, \dots, \underline{ym-1}, \underline{D2}[\underline{ym}], \underline{ym+1}, \dots, \underline{yn})$

The secondary transformator, $\underline{X2}[t]$:

- (9) :3: =: $s[rt]$
 ((10) :6: =: $\underline{?}$
 (11) :5: =: $f(\underline{y1}, \dots, \underline{ym-1}, \underline{D2}[\underline{ym}], \underline{ym+1}, \dots, \underline{yn})$

Remark 4.4

The normal evaluation takes the leftmost outermost tree for which a prefix exists for which a transition of $\underline{M}$ is applicable - and performs the transformation. As in [def. 4.3] computation bounces back to outermost containing interior provided the transformation of such were kept suspended by (5). The purpose of the outer $\underline{D1}$ in (2) is to push evaluation in direction left-to-right - upon completion of evaluation of $\underline{a}[\underline{ym}]$. The purpose of $\underline{X1}$ in (5) is to get $\underline{X1}$ applied to t as soon as evaluation of $\underline{a}[\underline{ym}]$ has completed.

Example 4.5 Leftmost Innermost Fevaluator, Vli (first version)

The distributor, D[t] (= meaning) :

- (0) :0: =: t
- (1) :8: =: X[t]
- (2) :5: =: D[f(y₁, ..., y_{m-1}, D[y_m], y_{m+1}, ..., y_n)]
- (3) :7: =: D[e(y₁, ..., y_{m-1}, D[y_m], y_{m+1}, ..., y_n)]

The transformator, X[t] :

- (4) :3: =: D[s[rt]]
- ((5) :4: =: ?)

Remark 4.5

In this evaluator no bouncing occurs - and thus only one level of distributor and transformator is apparently required. The outer D in (2) serves the purpose of finally getting X applied to t. The outer D in both (2) and (3) serves the purpose of getting evaluation to progress in direction left-to-right over disjoint trees.

Remark 4.3:5

The (#) rules in [defs. 4.3-5] take a tree in T(E+F') and transform

them into trees over the same domain. A computation on t in $\underline{T}(\underline{E})$ gets underway by applying $\underline{f_0} \in \underline{F}$ to t , $t' = \underline{f_0}(t)$, and a specific course of evaluation is fixed by selecting some $\underline{v}$ and applying meaning from $\underline{F'}$ to t' , $t'' = \text{meaning}[\underline{f_0}(t)]$. Observe: $t \in \underline{T}(\underline{E})$, $t' \in \underline{T}(\underline{E} + \underline{F}) / \underline{T}(\underline{E})$, $t'' \in \underline{T}(\underline{E} + \underline{F'}) / \underline{T}(\underline{E} + \underline{F})$.

We have used two types of grouping delimiters: "(" and ")", respectively: "[" and "]". The former type to denote - vaguely speaking - that part of t'' coming from $\underline{T}(\underline{E} + \underline{F})$, the latter type to single out whenever an $\underline{f'} \in \underline{F'}$ is present - or being introduced, in either case: by originating from $\underline{M'}$. The two types of delimiters of course have the same significance; one could indeed use just one of the pairs exclusively.

The evaluators were defined to cope deterministically with trees over $\underline{T}(\underline{E} + \underline{F})$. And now - what do we get, after a few application of rules (#) in one of [defs. 4.3-5], but a new tree, in an extended domain: $\underline{T}(\underline{E} + \underline{F'})$! Will this regression ever 'stop'? What is the nature of the 'evaluator-evaluator' which computes these new trees? How is the 'evaluator-evaluator' $\underline{VV}$ defined, does it resemble any of the already presented $(\underline{Vco}, \underline{Vn}, \underline{Vli})$? Section 5 attempts to provide some answers to these and related questions. A more complete treatment will appear in section 9 (part II).

5. DETERMINATE EVALUATIONS

This section will introduce a number of very problem oriented definitions whose purpose is to serve in the characterization of various aspects of FSTT's, FSTE's and the computations they invoke. Some of these definitions, especially [Def. 5.4], should ideally be followed up by a precise statement (lemma, theorem) about the decidability of satisfiability of the definitions to any arbitrarily given FSTT or FSTE. We have chosen to postpone such an analytical investigation of some of the properties of our systems, and hope to present such material in a subsequent part II. Instead this part I concentrates on the synthesis aspects, providing a suitable framework, demonstrating its applicability and giving constructions.

Definition 5.1 Deterministic FSTT

Let $\underline{A}$ be some FSTT. Let $t \in \underline{T}(\underline{E}+\underline{F})$ be some tree in some computation over $\underline{A}$. Let f be some symbol in $\underline{F}$ in t . f then is the top of some suffix subtree t' . Let $\underline{pt}_1$ and $\underline{pt}_2$ be two distinct prefixes in $\underline{prefix}[t']$. If $\underline{M}$ defines two (or more) fstt's:

$[\underline{lt}_1 =: \underline{rt}_1]$

$[\underline{lt}_2 =: \underline{rt}_2]$

for which $pt_1 = lt_1$ and $pt_2 = lt_2$, then A is said to be nondeterministic; if no such pair of fstt's exists in a given FSTT, then the FSTT is deterministic.

Remark 5.1

There are two basic ways in which non-determinism comes about:

- (1) Containment: Either $lt_1 \in \text{prefix}[lt_2]$; lt_1 is contained in lt_2 ; lt_2 'over-shadows' lt_1 . lt_1 .
- (2) Adjacency: Or there exist a 'maximal' prefix pt of both lt_1 and lt_2 such that $\text{suffix}[lt_1, pt]$ and $\text{suffix}[lt_2, pt]$ define exactly those parts of lt_1 and lt_2 not common to the other, i.e. for which the common parts consist of single symbols. lt_1 and lt_2 are then said to be adjacent.

From this we see that it is easy to check whether or not a given FSTT is deterministic.

Definition 5.2 Reduced FSTT

Let A be deterministic. Let $[lt_1=rt], \dots, [lt_n=rt]$ be a subset of M . Let $nt_i = n-p[lt_i]$, i.e. nt_i is usually a set of trees (in $T(E+F)$). Let $lt \in \text{prefix}[lt_i]$ for all i . Let $st_i = \text{suffix}[nt_i, lt]$ and $ST = \text{set-union}(i) st_i$. Let $h = \max (\text{height}[t] \text{ such that } t \in ST)$.

Then if $\underline{ST} = \underline{T}(\underline{E}+\underline{F}, \emptyset, \underline{top}[\underline{ST}]; h)$, then $\underline{ST}$ exhausts all possible trees no higher than h . If and only if this is the case, then we say that $\underline{A}$ is overspecified, and the above n fstt's can be reduced to one fstt of the form: $[\underline{lt} = \underline{rt}]$. $\underline{A}$ is said to be reduced if no such reduction is possible.

Again it is easy to verify whether a given FSTT is overspecified or reduced.

Definition 5.3 Partial and Total FSTT

If there exists some $t \in \underline{T}(\underline{E}+\underline{F})$ with a suffix tree $\underline{st} \in \underline{suffix}[t]$, $\underline{st} \in \underline{fT}(\underline{E}+\underline{F})$, such that for no fstt in $\underline{M}$ is there a prefix of $\underline{st}$, $\underline{pst}$, such that $\underline{pst} = \underline{lt}$ for that fstt, then the FSTT is a partial FSTT. If (instead) for all such pairs: $(t, \underline{st})$ there is some fstt in $\underline{M}$ such that $\underline{lt}$ of that fstt is a prefix of $\underline{st}$, then the FSTT is a total FSTT.

Lemma 5.3

Let $\underline{A}$ be deterministic and reduced. Let $\underline{S}$ range over all assignment-substitutions from $\underline{Z}$ to $\underline{T}(\underline{E}+\underline{F})$, and let $\underline{M} = ([\underline{lt}_1 = \underline{rt}_1], \dots, [\underline{lt}_n = \underline{rt}_n])$. Let $\underline{T}_k = \{\underline{s}[\underline{lt}_k] \mid \underline{s} \in \underline{S}\}$. Let $\underline{h}_i = \underline{height}[\underline{lt}_i]$. Then, if for each $t \in \underline{T}(\underline{E}+\underline{F}, \emptyset, \underline{F}; \underline{h}_i)$, t is also in the prefix: $\underline{prefix}[\underline{T}_k]$ for some k , then $\underline{A}$ is a total FSTT, otherwise $\underline{A}$ is partial.

Remark 5.1:3

Note that all of the above definitions apply to FSTTs and FSTEs (to the latter by virtue of an FSTE having an FSTT as its major component). The next three definitions apply to the computations defined by FSTTs.

Definition 5.4 (a) Determinate, (b) Serial
and (c) Free Computations

Let $\underline{A}$ be some FSTT, and $t \in \underline{T}(\underline{E}+\underline{F})$ be some step in a computation defined by $\underline{A}$. Let $p \in \text{path}[t]$. (a) $\underline{A}$ is said to define a determinate computation if along any such path p in t at most one symbol $f \in \underline{F}$ defines an applicable interior. (b) $\underline{A}$ defines a serial computation if for any such t there exist at most one applicable interior. (c) $\underline{A}$ defines a free computation if in any such nonconstant t there is at least one applicable interior; a tree $t \in \underline{T}(\underline{E}+\underline{F})$ with at least one f in the interior but no applicable interiors is blocked.

Remark 5.4

We shall now redefine the three inspector schemes of section 4. The individual redefinitions, $\underline{V}$, assume that the 'evaluator-evaluator', $\underline{VV}$, which in turn carries out the $\underline{V}$ evaluations, affords a determinate computation over $f' \in \underline{F}'$. In addition to determinacy the two latter evaluators, $\underline{V}_n$ and $\underline{V}_{li}$, furthermore effects serial

computations. Finally we wish to insure that if blocking occurs, that it is due not to $\underline{V}$ but to the FSTT, $\underline{A}$, embedded in $\underline{V}$ ($\underline{V}[\underline{A}]$). To achieve these ends the FSTT part of the $\underline{V}$'s will be (required to be) deterministic, reduced and partial.

Example 5.5 Second version of $\underline{V}_{co}$ [def. 4.3]

- | | | | |
|------|--|------|--|
| (0) | $\underline{meaning}[\underline{f0}(\underline{y1})]$ | $=:$ | $\underline{meaning}[\underline{D1}[\underline{f0}(\underline{y1})]]$ |
| (1) | $\underline{meaning}[\underline{D1c}[\underline{y1}]]$ | $=:$ | $\underline{y1}$ |
| (2) | $\underline{D1}[e()]$ | $=:$ | $\underline{D1c}[e()]$ |
| (3) | $\underline{D1}[e(\underline{y1}, \dots, \underline{yn})]$ | $=:$ | $\underline{Q1}[e(\underline{D1}[\underline{y1}], \dots, \underline{D1}[\underline{yn}])]$ |
| (4) | $\underline{D1}[f(\underline{y1}, \dots, \underline{yn})]$ | $=:$ | $\underline{X1}[f(\underline{y1}, \dots, \underline{yn})]$ |
| (5) | :3: $\underline{X1}[t]$ | $=:$ | $\underline{D1}[\underline{s}[rt]]$ |
| (6) | :6: $\underline{X1}[t]$ | $=:$ | $\underline{D1c}[?]$ |
| (7) | :5: $\underline{X1}[f(\underline{y1}, \dots, \underline{yn})]$ | $=:$ | $\underline{Q2}[f(\underline{D2}[\underline{y1}], \dots, \underline{D2}[\underline{yn}])]$ |
| (8) | $\underline{D2}[e()]$ | $=:$ | $\underline{D2c}[e()]$ |
| (9) | $\underline{D2}[e(\underline{y1}, \dots, \underline{yn})]$ | $=:$ | $\underline{Q1}[e(\underline{D2}[\underline{y1}], \dots, \underline{D2}[\underline{yn}])]$ |
| (10) | $\underline{D2}[f(\underline{y1}, \dots, \underline{yn})]$ | $=:$ | $\underline{X2}[f(\underline{y1}, \dots, \underline{yn})]$ |
| (11) | :3: $\underline{X2}[t]$ | $=:$ | $\underline{D2}[\underline{s}[rt]]$ |
| (12) | :6: $\underline{X2}[t]$ | $=:$ | $\underline{D2c}[?]$ |
| (13) | :5: $\underline{X2}[f(\underline{y1}, \dots, \underline{yn})]$ | $=:$ | $\underline{Q1}[f(\underline{D2}[\underline{y1}], \dots, \underline{D2}[\underline{yn}])]$ |
| (14) | $\underline{Q1}[e(\underline{D1c}[\underline{y1}], \dots, \underline{D1c}[\underline{yn}])]$ | $=:$ | $\underline{D1c}[e(\underline{y1}, \dots, \underline{yn})]$ |

$$(15) \quad \underline{Q1}[u(\underline{D2c}[\underline{y1}], \dots, \underline{D2c}[\underline{yn}])] \quad =: \quad \underline{D2c}[u(\underline{y1}, \dots, \underline{yn})]$$

$$(16) \quad \underline{Q2}[f(\underline{D2c}[\underline{y1}], \dots, \underline{D2c}[\underline{yn}])] \quad =: \quad \underline{X1}[f(\underline{y1}, \dots, \underline{yn})]$$

Remark

The above definition specifies that:

$$\forall \underline{\text{meaning}}, \underline{D1}, \underline{D2}, \underline{X1}, \underline{X2}, \underline{Q1}, \underline{Q2} \quad \forall \underline{F} \quad \underline{\text{in}} = \underline{F'}$$

$$\forall \underline{D1c}, \underline{D2c} \quad \forall \underline{E} \quad \underline{\text{in}} = \underline{E'}$$

Example 5.6 Second version of $\underline{Vn}$ [def. 4.4]

In the appendix, section IV, we graphically illustrate the next 31 fstt's as a set of 31 pairs of (2 dimensional) trees. This indeed represents the way in which we personally 'program' in this 'language' of tree transformations. We believe it to be quite natural. From the graphical representation it is then easy to mechanically generate the linearization now given:

$$(0) \quad \underline{\text{meaning}}[\underline{f0}(\underline{y1})] \quad =: \quad \underline{\text{meaning}}[\underline{D1}[\underline{f0}(\underline{y1})]]$$

$$(1) \quad \underline{\text{meaning}}[\underline{D1c}[\underline{y1}]] \quad =: \quad \underline{y1}$$

$$(2) \quad \underline{D1}[\underline{f1}] \quad =: \quad \underline{X1}[\underline{f1}]$$

$$(3) \quad \underline{D1}[\underline{e}()] \quad =: \quad \underline{D1c}[\underline{e}()]$$

$$(4) \quad \underline{D1}[\underline{e}(\underline{y1}, \dots, \underline{yn})] \quad =: \quad \underline{Q1}[\underline{e}(\underline{Q2}[\underline{y1}, \dots, \underline{yn}])]$$

- (5) $\underline{Q2}[\underline{y1}] =: \underline{Q3}[\underline{D1}[\underline{y1}]]$
- (6) $\underline{Q2}[\underline{y1}, \dots, \underline{yn}] =: \underline{Q3}[\underline{D1}[\underline{y1}], \underline{S1}[\underline{y2}, \dots, \underline{yn}]]$
- (7) $\underline{Q3}[\underline{D1c}[\underline{y1}]] =: \underline{Q3}[\underline{y1}]$
- (8) $\underline{Q3}[\underline{D1c}[\underline{y1}], \underline{S1}[\underline{y2}, \dots, \underline{yn}]] =: \underline{Q3}[\underline{y1}, \underline{Q2}[\underline{y2}, \dots, \underline{yn}]]$
- (9) $\underline{Q3}[\underline{y1}, \underline{Q3}[\underline{y2}, \dots, \underline{yn}]] =: \underline{Q3}[\underline{y1}, \underline{y2}, \dots, \underline{yn}]$
- (10) $\underline{Q1}[e(\underline{Q3}[\underline{y1}, \dots, \underline{yn}])] =: \underline{D1c}[e(\underline{y1}, \dots, \underline{yn})]$
- (11) :3: $\underline{x1}[t] =: \underline{D1c}[\underline{s}[rt]]$
- (12) :5: $\underline{x1}[f(\underline{y1}, \dots, \underline{yn})] =: \underline{Q4}[f(\underline{Q5}[\underline{y1}, \dots, \underline{yn}])]$
- ((13) $\underline{Q4}[f(\underline{S3}[\underline{y1}, \dots, \underline{yn}])] =: \underline{D1c}[\underline{?}]$
- (14) $\underline{Q4}[f(\underline{S2}[\underline{y1}, \dots, \underline{yn}])] =: \underline{x1}[f(\underline{y1}, \dots, \underline{yn})]$
- (15) $\underline{Q5}[\underline{y1}] =: \underline{Q5}[\underline{D2}[\underline{y1}]]$
- (16) $\underline{Q5}[\underline{y1}, \dots, \underline{yn}] =: \underline{Q5}[\underline{D2}[\underline{y1}], \underline{S1}[\underline{y2}, \dots, \underline{yn}]]$
- (17) $\underline{Q5}[\underline{D2e}[\underline{y1}]] =: \underline{S2}[\underline{y1}]$
- (18) $\underline{Q5}[\underline{D2f}[\underline{y1}]] =: \underline{S3}[\underline{y1}]$
- (19) $\underline{Q5}[\underline{D2e}[\underline{y1}], \underline{S1}[\underline{y2}, \dots, \underline{yn}]] =: \underline{Q5}[\underline{y1}, \underline{Q5}[\underline{y2}, \dots, \underline{yn}]]$
- (20) $\underline{Q5}[\underline{D2f}[\underline{y1}], \underline{S1}[\underline{y2}, \dots, \underline{yn}]] =: \underline{S2}[\underline{y1}, \underline{y2}, \dots, \underline{yn}]$
- (21) $\underline{Q5}[\underline{y1}, \underline{S2}[\underline{y1}, \dots, \underline{yn}]] =: \underline{S2}[\underline{y1}, \underline{y2}, \dots, \underline{yn}]$
- (22) $\underline{Q5}[\underline{y1}, \underline{S3}[\underline{y2}, \dots, \underline{yn}]] =: \underline{S3}[\underline{y1}, \underline{y2}, \dots, \underline{yn}]$
- (23) $\underline{D2}[e()] =: \underline{D2e}[e()]$
- (24) $\underline{D2}[\underline{f1}] =: \underline{x2}[\underline{f1}]$
- (25) $\underline{D2}[e(\underline{y1}, \dots, \underline{yn})] =: \underline{Q6}[e(\underline{Q5}[\underline{y1}, \dots, \underline{yn}])]$

- (26) $\underline{Q6}[u(\underline{S2}[\underline{y1}, \dots, \underline{yn}])] \quad =: \quad \underline{D2f}[u(\underline{y1}, \dots, \underline{yn})]$
 (27) $\underline{Q6}[u(\underline{S3}[\underline{y1}, \dots, \underline{yn}])] \quad =: \quad \underline{D2e}[u(\underline{y1}, \dots, \underline{yn})]$
- (28) :3: $\underline{X2}[t] \quad =: \quad \underline{D2f}[\underline{s}[rt]]$
 (29) :5: $\underline{X2}[f(\underline{y1}, \dots, \underline{yn})] \quad =: \quad \underline{Q6}[f(\underline{Q5}[\underline{y1}, \dots, \underline{yn}])]$

Remark 5.6

The above definition specifies that:

$$\underline{E} + \forall \underline{D1c}, \underline{D2e}, \underline{D2f}, \underline{S1}, \underline{S2}, \underline{S3} \text{ ' in= } \underline{E}'$$

$$\forall \underline{\text{meaning}}, \underline{D1}, \underline{D2}, \underline{X1}, \underline{X2}, \underline{Q1}, \underline{Q2}, \underline{Q3}, \underline{Q4}, \underline{Q5}, \underline{Q6} \text{ ' } + \underline{F} \text{ in= } \underline{F}'$$

fstt (5) (and (15)) takes precedence over fstt (6) (and (16)). That is, the rank of $\underline{Q2}$ in (5) is 1, in (6) n is larger than 1; and likewise for $\underline{Q5}$ in (15) and (16). We shall subsequently extend our definition of FSTT's to cope with 'ordered' sets of fstt's.

Example 5.7 Second version of $\underline{vli}$ [def. 4.5]

- (0) $\underline{\text{meaning}}[\underline{f0}(\underline{y1})] \quad =: \quad \underline{\text{meaning}}[\underline{D}[\underline{f0}(\underline{y1})]]$
 (1) $\underline{\text{meaning}}[\underline{Dc}[\underline{y1}]] \quad =: \quad \underline{y1}$
- (2) $\underline{D}[e()] \quad =: \quad \underline{Dc}[e()]$
 (3) $\underline{D}[u(\underline{y1}, \dots, \underline{yn})] \quad =: \quad \underline{Q1}[u(\underline{Q2}[\underline{y1}, \dots, \underline{yn}])]$

- (4) $\underline{Q1}[e(\underline{S2}[\underline{e1}, \dots, \underline{en}])] =: \underline{Dc}[e(\underline{e1}, \dots, \underline{en})]$
- (5) $\underline{Q1}[f(\underline{S2}[\underline{y1}, \dots, \underline{yn}])] =: \underline{X}[f(\underline{y1}, \dots, \underline{yn})]$
- (6) $\underline{Q2}[\underline{y1}] =: \underline{Q2}[\underline{D}[\underline{y1}]]$
- (7) $\underline{Q2}[\underline{y1}, \dots, \underline{yn}] =: \underline{Q2}[\underline{D}[\underline{y1}], \underline{S1}[\underline{y2}, \dots, \underline{yn}]]$
- (8) $\underline{Q2}[\underline{Dc}[\underline{y1}], \underline{S1}[\underline{y2}, \dots, \underline{yn}]] =: \underline{Q2}[\underline{y1}, \underline{Q2}[\underline{y2}, \dots, \underline{yn}]]$
- (9) $\underline{Q2}[\underline{Dc}[\underline{y1}]] =: \underline{S2}[\underline{y1}]$
- (10) $\underline{Q2}[\underline{y1}, \underline{S2}[\underline{y2}, \dots, \underline{yn}]] =: \underline{S2}[\underline{y1}, \underline{y2}, \dots, \underline{yn}]$
- (11) :3: $\underline{X}[\underline{t}] =: \underline{D}[\underline{s}[\underline{rt}]]$
- ((12) :4: $\underline{X}[\underline{t}] =: \underline{Dc}[\underline{?}]$

Remark 5.5:7

In [defs. 5.5-7] fstt's (0) and (1) respectively initialize and terminate computations. We shall subsequently prove (in part II) that if in (0) $\underline{s1}[\underline{y1}]$ is in $\underline{T}(\underline{E})$, then in (1) - whenever reached - $\underline{s2}[\underline{y1}]$ will also be in $\underline{T}(\underline{E})$; here $\underline{s1}$ and $\underline{s2}$ are the substitutions defined by the particular assignments.

Completion 5.8

To 'complete' an FSTE $\underline{B}$ given an FSTT $\underline{A}$ delete rules [5.5(5,11)], [5.6(11,28)] and [5.7(11)] and for each fstt: $[\underline{lti} =: \underline{rti}]$ in $\underline{A}$ add to $\underline{B}$ fstt's of the type: $\underline{X}[\underline{lti}] =: \underline{D}[\underline{s}[\underline{rti}]]$, where $\underline{X}$ and $\underline{D}$ (if the latter is present) are the $\underline{X}$'s and $\underline{D}$'s of the deleted 'rules'. Delete fstt's: [5.5((6),(12))] and [5.7((12))]. Let $\underline{lt}' =: \underline{rt}'$ range

over a (finite) set, $\underline{A}'$, of fstt's such that together with $\underline{A}$, $\underline{A}'+\underline{A}$ form a total FSTT. And finally delete fstt's: [5.5(7,13)] and [5.6(12,29)]. In lieu of the latter and in the corresponding places insert a set of fstt's: $\underline{X}[\underline{1t}'] =: \underline{Q}[f(\underline{D}[\underline{t1}, \dots, \underline{tn}])]$ where each $\underline{1t}' = f(\underline{t1}, \dots, \underline{tn})$. $\underline{Q}$ and $\underline{D}$ range over respectively the pairs: $\underline{Q2}$, $\underline{D2}$; $\underline{Q1}$, $\underline{D2}$; $\underline{Q4}$, $\underline{Q5}$ and $\underline{Q6}$, $\underline{Q5}$.

This construction takes a part of the completed FSTE and turns it into a total FSTT; this is required if the FSTE shall have any chance of deterministically deciding whether an fstt (in $\underline{A}$) applies, or whether to proceed further up the tree. Notice that the remaining part of the FSTE, seen as an FSTT, is still only partial.

6. TREE AUTOMATA

We now relate the FSTT's to the theory of tree automata. This section essentially provides an overview, whereas section 8. presents our contribution to the constructive aspects of the theory (applied tree automata !).

6.1 Review of Tree Automata

We shall briefly review some results from the theory of tree automata. From Thatcher, [Ref. 16] we lift:

[Def. 6.1.1] An E-algebra is a pair: $\underline{Ag} = [\underline{S}, \underline{g}]$, where $\underline{S}$ is a finite set, called the 'carrier' of $\underline{Ag}$, and $\underline{g}$ assigns to each $e \in \underline{E}$ a function:

$$\underline{g}_e : \underline{S}_e =: \underline{S}, \underline{S}_e = \underline{\text{set-union}}(r(e,n)) \underline{S}^{**n},$$

where $\underline{X}^{**m}$ denotes the cartesian product of m factors (sets) $\underline{X}$.

[Def. 6.1.2] Let $\underline{Ag}$ be an E-algebra and $\underline{As} :$

$$\underline{As} : \underline{Z} =: \underline{S}$$

be an assignment of values in $\underline{S}$ to the variables in $\underline{Z}$.

[Def. 6.1.3] The assignment $\underline{As}$ determines a unique homomorphism from $\underline{T}(\underline{E}, \underline{Z})$ into $\underline{Ag}$:

- $$\begin{aligned}
 (0) \quad & \underline{hAs}(d) = \text{if } d \in \underline{Z} \\
 & \quad \text{then } \underline{As}(d) \\
 & \quad \text{else} \\
 & \quad \text{if } d \in \underline{E0} \\
 & \quad \text{then } \underline{gd}() \\
 (1) \quad & \underline{hAs}(e(\underline{t1}, \dots, \underline{tn})) = \underline{ge}(\underline{hAs}(\underline{t1}), \dots, \underline{hAs}(\underline{tn}))
 \end{aligned}$$

If $t \in \underline{T}(\underline{E})$ then the image of t under $\underline{hAs}$ is independent of $\underline{As}$; we will write $\underline{hAg}$ to denote this unique homomorphism of $\underline{T}(\underline{E})$ into $\underline{Ag}$.

[Def. 6.1.4] A finite E-automaton, $\underline{M}$, is a pair: $[\underline{Ag}, \underline{F}]$, where $\underline{Ag} = [\underline{S}, \underline{g}]$ is a finite $\underline{E}$ -algebra and $\underline{F}$, the final states, is a subset of $\underline{S}$.

[Def. 6.1.5] A constant $\underline{E}$ -expression t is recognized by $\underline{M}$ if $\underline{hAg}(t) \in \underline{F}$. [Def. 6.1.6] The set, $\underline{B}(\underline{M})$, of $\underline{E}$ -expressions recognized by $\underline{M}$ is:

$$\underline{B}(\underline{M}) = \{ t \mid \underline{hAg}(t) \in \underline{F} \}$$

This set of terse definitions has the following mechanical interpretation: Let t be a constant E -expression. Then, in order to check whether t is recognized by M , one 'affixes' the operator hAg to the root of t , $top[t]$. One can then go on explaining the effect of doing so by saying that the operator hAg 'moves' up the tree, from root to frontier. Each move bring (the) hAg ('s) one level further away from $top[t]$, affixing copies of hAg 's to proper subtree-roots. In making such a move, hAg assigns to the node e which it is leaving a function ge . In reaching the leaves hAg assigns to these some similar functions $gd_1, \dots, gd_l$. The last hAg leaves the tree through the leaves! And now the direction of action reverses, and moves from the frontier toward the root under influence of g . First gd assign states $s \in S$ to leaves of t (whose nodes by now are all relabelled, so that only the original structure is preserved; so we ought not to call it t , but something else). Let (in general) ge be the direct ancestor (node label) of a tuple of n such leaves $[s_{i1}, \dots, s_{in}]$; g then effects the assignment of a state s_j to node ge , while at the same time discarding (pruning) its direct descendants. And so forth. Finally the tree collapses to a single node labelled with some s in S . If $s \in F$, then the original tree t has been recognized by M .

Taking a somewhat simpler set of definitions leading up to finite state tree automata and recognizability is certainly possible [Ref. 15]: Omitting the first phase, the 'root-to-frontier' movement, we could (directly) have assigned to all leaves, d , states according

to $\underline{g_d}$, and then defined a set of transitions rules:

$$e[\underline{s_{i1}}[\underline{y_1}], \dots, \underline{s_{in}}[\underline{y_n}]] := \underline{s_j}$$

where $e \in \underline{E}$, $\underline{r}[e, n]$ holds, $\underline{y_k}$ ranges over all $[e, k] \in \underline{Z} = \underline{E}^* \underline{I}$ and the states $\underline{s_{i1}}, \dots, \underline{s_{in}}, \underline{s_j} \in \underline{S}$. Actually, one could keep the tree parts which are being pruned by letting the right part of the above fstd be: $\underline{s_j}[e[\underline{y_1}, \dots, \underline{y_n}]]$. This set of transition rules would in either case not be quite as implied above by $\underline{g_e}$. What we would have gotten, however, would be the so-called 'frontier-to-root' automaton, $\underline{FRM}$, as contrasted to the so-called 'root-to-frontier' automaton, $\underline{RFM}$. The $\underline{FRM}$ will have recognized a tree if the final state assigned to the root is in $\underline{F}$. The $\underline{RFM}$ will have transition rules of the form:

$$\underline{s_i}[e[\underline{y_1}, \dots, \underline{y_n}]] := e[\underline{s_{k1}}[\underline{y_1}], \dots, \underline{s_{kn}}[\underline{y_n}]]$$

where $\underline{r}[e, n]$ holds, $e, \underline{e_{j1}}, \dots, \underline{e_{jn}} \in \underline{E}$, $\underline{s_i}, \underline{s_{k1}}, \dots, \underline{s_{kn}} \in \underline{S}$ and as before: $\underline{y_j}$ ranges over all $[e, j]$ in $\underline{Z} = \underline{E}^* \underline{I}$. Such an automaton would be initialized at the root in some initial state, and would have recognized a tree if and only if all leaves are finally assigned states all in $\underline{F}$. All the reviewed automata are of the Moore type in that we associate recognition with final states rather than output -- in fact we have not even bothered to define such an output.

From the above discussion it should be clear that the kind of

finite state transitions we have defined in the preceding sections amount to some different kind of automata (machines). Firstly: We allow that there be any (finite) number of states along any one path in these trees including none at all. This is then in contrast to the above tree automata which have exactly one state associated with each path of the tree being recognized. Secondly: We are out, not just to recognize trees, but also to transform them while recognizing (subtrees of) trees. (Incidentally: Where in sections 2-5 the states all were in $\underline{F}$, that $\underline{F}$ is here $\underline{S}$. The set of final states of the tree automata is called $\underline{F}$ but has otherwise no relation to the former $\underline{F}$!).

The combined root-to-frontier followed by the frontier-to-root movement inherent in definitions 6.1.3 and 6.1.1 will later be seen to properly reflect what goes on in our 'realizations' as well as it properly 'models' some of the mechanisms we came up with for the various types of evaluators.

[Def. 6.1.7] Let $[\underline{E}, \underline{r}]$ be a ranked alphabet, and define:

$$\underline{R} = \{[e, w] \mid w \in \text{clos}[\underline{E}] \ \& \ \underline{r}[e, \text{length}[w]]\}$$

The structure $\underline{G} = [\underline{P}, \underline{\lambda}]$, where $\underline{P} \text{ in } \underline{R}$ and $\underline{\lambda} \in \underline{E}/\underline{E}_0$, is a context-free grammar. $\underline{P}$ is the finite set of 'productions' of $\underline{G}$. $\underline{\lambda}$ is the 'initial symbol'.

[Def. 6.1.8] The set, $\underline{D}(\underline{G}, e)$ of e-phrase-markers is defined by

the following simultaneous inductive definition:

if $[e, e_1 \dots e_n] \in P$ and $t_i \in D[G, e_i] \quad (i=1, \dots, n)$
 then $e(t_1, \dots, t_n) \in D[G, e]$

The subset of $T(E)$ generated by G is $T(G) = D[G, X]$ and the language [Def. 6.1.9] generated by e is $L(G, e) = \text{yield}[D[G, e]]$. $L(G, X) = L(G)$.

[Def. 6.1.10] A Subset T' of $T(E)$ will be called local if there exists a context-free grammar G over E such that $T' = T(G)$. [Def. 6.1.11] A language $L' \subseteq \text{clos}[E]$ is context-free if $L' = L(G)$ for some context-free grammar G .

[Lemma 6.1.12] Local subsets of $T(E)$ are recognizable.

[Theorem 6.1.13] $L \subseteq \text{clos}[E]$ is context-free if and only if $L = \text{yield}[T]$ for some recognizable subset $T \subseteq T(E)$.

6.2 Discussion

In a certain sense the theory of tree automata is 'clean' in that it on one hand is concerned with finite state tree automata which are capable of recognizing context-free sets of trees -- while on the other hand staying clear of getting involved in the terrible technical details of getting funny looking and weirdly behaving

pushdown stacks moving up and down, pushing and popping all kinds of strangely shaped symbols to and from the stack. If we started our involvement in context-free language recognition and translation using finite state tree automata rather than push-down transducers, then perhaps we could arrive at a more intrinsic approach -- from where we could then develop the technical subissue of mapping such tree automata onto push-down machines.

The set of possible FSTTs and the set of constant trees in $\underline{T(E)}$ can now be viewed as (yet another) programming language. Before doing so we shall first postulate a general frame for languages and their meaning. Then we shall show how our language fit into that frame. From that we shall go on to define 'realizations' of such programming languages, specifically their interpreter. The realization will be tied to restricted -- although still completely general -- tree systems. We shall therefore need yet a number of definitions leading us up to context-free tree programming languages and their realization in terms of yet a third class of still very simple finite state tree automata.

In part II we shall bring a more complete treatment of classes of tree automata and a more direct comparison of our work to that of e.g. Thatcher and Rounds [Ref. 18, 14].

the first of these is the fact that the
 the second is the fact that the
 the third is the fact that the
 the fourth is the fact that the
 the fifth is the fact that the
 the sixth is the fact that the
 the seventh is the fact that the
 the eighth is the fact that the
 the ninth is the fact that the
 the tenth is the fact that the

the first of these is the fact that the
 the second is the fact that the
 the third is the fact that the
 the fourth is the fact that the
 the fifth is the fact that the
 the sixth is the fact that the
 the seventh is the fact that the
 the eighth is the fact that the
 the ninth is the fact that the
 the tenth is the fact that the
 the first of these is the fact that the
 the second is the fact that the
 the third is the fact that the
 the fourth is the fact that the
 the fifth is the fact that the
 the sixth is the fact that the
 the seventh is the fact that the
 the eighth is the fact that the
 the ninth is the fact that the
 the tenth is the fact that the

the first of these is the fact that the
 the second is the fact that the
 the third is the fact that the
 the fourth is the fact that the
 the fifth is the fact that the
 the sixth is the fact that the
 the seventh is the fact that the
 the eighth is the fact that the
 the ninth is the fact that the
 the tenth is the fact that the
 the first of these is the fact that the
 the second is the fact that the
 the third is the fact that the
 the fourth is the fact that the
 the fifth is the fact that the
 the sixth is the fact that the
 the seventh is the fact that the
 the eighth is the fact that the
 the ninth is the fact that the
 the tenth is the fact that the

7. CONTEXT FREE TREE PROGRAMMING

The present section starts out by making some remarks concerning what constitutes a programming language. These remarks do not advance a particular viewpoint, but attempts to accomodate several within their frame of reference. We then go on to propose a restriction of sets, FSTT, of fstt's, allowing for these to be ordered. We comment on the (seeming) relations to e.g. LISP [Ref. 12] conditional expressions, and MARKOV Normal Algorithms [Ref. 11]. We finally show how one can render the communication (writing and reading) of fstt's more palatable. Essentially what we have in mind is to restrict tree transformations to local sets, i.e. trees generated by some context-free grammar. The fstt's will therefore "to some degree" also be generable by some context-free grammar, and we can consequently reduce the specification of an fstt to that of its operators, their position in the (term) trees, and the frontier, yield, of these trees. That part of the section which introduces such (term) tree short-hand notations is of less importance compared to the part which advances the notion of an ordered set of fstt's.

7.1 Languages and Meaning

Let a language be rigorously but informally defined as having three parts: two sets and a mapping.

The first set is the set, $\underline{L}$, of labels. The second set is the universe of discourse $\underline{Q}$. A label $\underline{l}$ in $\underline{L}$ play the role of naming a subset $\underline{q}$ in $\underline{Q}$. The mapping: meaning, applies to labels and yields subsets $\underline{q}$. Let q denote a generic element of $\underline{Q}$ ($\underline{q}$).

A label is either simple or composite. A composite label is the composition of simple and composite labels. The problem of defining meaning is from the knowledge of the meaning of simple labels to devise a procedure which computes the meaning of composite labels.

7.2 Tree Programming

We have in the past defined and spoken of the meaning of a (constant) tree given an FSTT $\underline{A}$ and an FSTE $\underline{V}$. We could as well speak of the meaning of an FSTT relative to some set of trees and an FSTE. This will be the viewpoint prevailing in the subsequent discussion (2).

There are basically three (equivalent) ways of describing our tree programming language from the viewpoint established in 7.1.

- (1) $\underline{L}$ is the set of all FSTTs, the mapping applies to an FSTT and selects a subset of $\underline{U}$, which is a set of 'doublets'. Each doublet has as its first member some tree, t , in $\underline{T(E)}$, and as its second member either (i) the atomic meta-symbol: meaningless, denoting that (some)

computation(s) on t does not terminate, or (ii) the symbol: $\underline{?}$, which may be defined as part of $\underline{E}$ and denotes undefined, which stands for some blocked computation, or (iii) the second member of the above doublet is a non-empty set of trees in $\underline{T(E)}$.

- (2) $\underline{L}$ is a set of doublets, the mapping is $\underline{V}$ as defined in section 4, and $\underline{U}$ is the set of trees $\underline{T(E)}$ augmented by at least the atomic meta-symbol: meaningless, and in case the atomic symbol: $\underline{?}$ was not part of $\underline{E}$, then also $\underline{?}$. Doublets of $\underline{L}$ are pairs of one (particular) FSTT paired with a tree t in $\underline{T(E)}$. The mapping from $\underline{L}$ to $\underline{U}$ selects either the nonterminating symbol, meaningless, the undefined symbol, $\underline{?}$, or some non-empty subset of $\underline{T(E)}$. This second explication of $\underline{L}$ can be changed (2') to allow for subsets of trees being the second member of the labels. The subset of $\underline{U}$ selected by the mapping is now of a different nature. It may include one or both of the 'special' atomic symbols mentioned above.

- (3) The label set is a pair of sets: the set of all FSTTs and the set $\underline{T(E)}$. The mapping takes one element from each, i.e. an $\underline{A}$ and a t , and selects a subset of $\underline{U}$ which is a defined in (2). Thus the mapping is very much like an FSTE $\underline{V}$ as defined previously.

If the mapping used in (2) and (3) is based on one of the

evaluators given in section 5, then the subsets selected will be singleton sets. If -- as briefly discussed in section 3.10 -- we allow non-constant, but still complete, trees as initial trees and blocked trees as the result of a computation, then in (1,2,3) above, where $\underline{?}$ appears, we would instead get (a set of) non-constant (complete) trees.

Definition 7.3 f.s.t. Programming Systems, Θ .

A f.s.t. programming language system, Θ , is a 3 tuple:

$$\Theta = [(1):[\underline{E}, \underline{F}, \underline{Z}], (2): \underline{V}, (3): [\underline{FSTT}, T(\underline{E})]]$$

where (1) denote a system for defining and the definition of alphabets $\underline{E}$ and $\underline{F}$ and the finite set $\underline{Z}$. From this we are then able to define trees in $T(\underline{E}+\underline{F}, \underline{Z})$ and fstt's. (2) $\underline{V}$ is a universal f-inspector scheme -- already, or presently being, defined. (3) denote a system for defining FSTT's in the set of $\underline{FSTT}$ s and subsets $T(\underline{E})$ of $T(\underline{E})$.

Definition 7.4 The Syntax of Θ programs

A program in Θ is a finite ordered set, $\underline{C}$, of fstt's. Viewed as an FSTT, $\underline{A}$, we shall require that $\underline{A} (= \underline{C})$ be proper.

Definition 7.5 The semantics of Θ programs

We shall now define the special notion of Markov applicability.

Let t be a tree in $T(\underline{E}+\underline{F})$ subject to transformation. Let st be some suffix tree in t with root $\underline{fi} = \underline{top}[st]$. Let all fstt's in $\underline{C}$ whose corresponding lt part has root $\underline{fi}$ be the ordered subset of $\underline{C}$:

$$[lt_{i1}=:rt_{i1}], \dots, [lt_{im}=:rt_{im}]$$

An fstt ($fstt_j =$) $[lt_{ik}=:rt_{ik}]$ for $1 \leq k \leq m$ is markov applicable to st if it is applicable in the normal sense of [Def. 3.2] and if for no j less than k there is an fstt ($fstt_j =$) $[lt_{ij}=:rt_{ij}]$ which is applicable to st in the normal sense of [Def. 3.2].

The meaning of a program $\underline{C}$ in Θ is derived from the completion of $\underline{V}$ with respect to $\underline{C}$. We shall subsequently define this new completion operation as a modification to the already informally defined $\underline{V}[\underline{A}]$ completion. Our modification consists in changing $\underline{C}$ into an unordered set, $\underline{A}$, of fstt's.

Remark 7.5

Let $\underline{C}$ be some program of ordered fstt's, and let $\underline{A}$ be the corresponding FSTT of unordered fstt's. Let $fstt_1$ precede $fstt_2$ in $\underline{C}$. The ordering of fstt's in $\underline{C}$ leads to the following properties of $\underline{C}$:

(1) $\underline{C}$ will always be deterministic. $\underline{A}$ may be non-deterministic, but

non-determinism will be removed because of the ordering:

- (2) Containment of f_{stt1} in f_{stt2} (or f_{stt2} in f_{stt1}) in $\underline{A}$ leads to f_{stt2} never being applied in any computation over $\underline{V}[\underline{C}]$ because f_{stt1} is inspected before f_{stt2} .
- (3) Adjacency of f_{stt1} and f_{stt2} leads to f_{stt1} and f_{stt2} being applied deterministically.
- (4-5) The concepts of (4) overspecified- and reduced- and (5) total- and partial- FSTT's $\underline{A}$ apply unchanged to programs $\underline{C}$.

Remark 7.3:5

A tree t in $T(\underline{E})$ denote an input to $\underline{C}$. $\underline{V}[\underline{C}]$ applied to t define the output, $\underline{V}[\underline{C}[t]]$, (if it is defined) from $\underline{C}$. For t ranging over $T(\underline{E})$ the output ranges accordingly over the universe of discourse which is: $\underline{U} = \text{"meaningless"} + ? + T(\underline{E})$.

Remark 7.6:7

Conventionally, programs are written as linear strings, w , presupposing some grammar, $\underline{G}$. With $\underline{G}$ known, to the compiler or interpreter, w is then parsed into a phrase-marker tree, t , whose yield is w . Then the compiler generates code according to the structure of t , or the interpreter starts transforming t , finally ending up with its meaning, t' , after which the interpreter outputs

the yield of t' as the result, usually presupposing some, possibly the same, grammar, G' .

The idea behind the next set of definitions is to permit the specification of FSTT's not as pairs of variable trees, but rather as pairs of ('nearly') linear strings. In these, the yield of the former tree fstt parts constitutes the major portion of these linear strings. We cannot directly take the yield and leave it with that; we must in some way indicate where operators, f , are to be positioned, and which operators are to be used. The rôle of the grammar is to give structure to the strings for the sake (1) of user convenience in knowing, a priori, which strings are well-formed, such that only these are stated, and (2) implementation convenience in providing for transparent and efficient realizations. If the allowable forms of fstt's were specified axiomatically, through general properties, then we do not in general have the benefit of the generative syntactic specification for which we today know quite a number of techniques leading up to speedy and economic implementations. Some of these techniques -- especially the ones we demonstrate in section 8. -- are very naturally modelled by actual hardware devices such as sequential machines.

Definition 7.6 Context-free Θ Systems, Θ'

A context-free f.s.t programming language system, Θ' , is a restriction on a system Θ in the following areas:

- (1) The set of input trees $T(\underline{E})$ is generated by some grammar $\underline{G}_i$ as the set of phrase marker trees $\underline{D}(\underline{G}_i, \lambda)$. So $T(\underline{E})$ is specified by giving $\underline{G}_i$.
- (2) The set of state trees $t \in \underline{T}(\underline{E}+\underline{F})$ is a subset of some set $\underline{D}(\underline{G}_s, \lambda')$ for some state grammar $\underline{G}_s$. Usually $\underline{G}_s$ can not be given explicitly. It can however be 'approximated' from the set of all programs $\underline{C}$. For any given program $\underline{C}$ one can derive a specific state grammar $\underline{G}_s'$. Normally the set of all trees t in $\underline{T}(\underline{E}+\underline{F})$ encountered in any computation based on $\underline{C}$ will be a recursively enumerable subset of $\underline{T}(\underline{E}+\underline{F})$.
- (3) $\underline{D}(\underline{G}_i, \lambda) \text{ in } \underline{D}(\underline{G}_s, \lambda')$
- (4) The set of output trees is left unspecified.
- (5) The form of the left- and right part of all fstt's is restricted to be context-free fstt's, see the next definition.

Definition 7.7 Context-free fstt

Let $\underline{G}_i$ define $\underline{E}_i$ and $\underline{G}_s$ define $\underline{E}_s$, then (1) $\underline{E}_i \text{ in } \underline{E}_s$. For any fstt: $\underline{lt} := \underline{rt} \in \underline{C}$, and any $\underline{it} \in \nabla \text{interior}[\underline{lt}] + \text{interior}[\underline{rt}] \nabla$ for which $\underline{it} \in \underline{T}(\underline{E}_s)$ we require that (2) $\underline{it} \in \text{interior}[\underline{D}(\underline{G}_s, \lambda')]$.

If $\underline{G_s}$ is explicitly given, then (2) can be loosened to require (2') that $\underline{n-p[lt]}, \underline{n-p[rt]} \in \underline{\text{interior}}[\underline{D(G_s, X')}]$.

Definition 7.8 Term tree fstt's

Let $\underline{G_i}$ be given and let $t \in \underline{T(E)}$. If $\underline{G_i}$ is unambiguous, then

$\underline{\text{top}[t]} \ [\ \underline{\text{yield}[t]} \]$

uniquely defines all of t . The brackets: " $\underline{[}$ " and " $\underline{]}$ " serve to delineate the scope of the root $\underline{\text{top}[t]}$ of the tree. In a 'linearized' fstt, or in an fstt 'term', as we shall come to call the desired fstt's, we will find that we in general must allow for the substitution of terms of the above kind in place of symbols of the string between $\underline{[}$ and $\underline{]}$. Not giving $\underline{G_s}$, but in general having to operate with non-constant complete trees, we then must find a way to precisely specify the presence of operators and yet avoid all the redundant information hitherto part of our context-free phrase-marker trees. We shall therefore presently define an operator which so-to-speak 'carves' out all the redundancy of a tree, leaving us with the bare essentials, the resulting string, will be called a term-tree.

Let $t \in \underline{T(U, Z)}$, let $\underline{G_i}$ be given, let $w \in \text{clos}[\underline{U+Z}]$ and $\underline{v_i} \in \underline{U+Z}$. Informally defined we then give the various cases of the term operator:

"if" $t \in \underline{U} + \underline{Z}$, "then" $\underline{term}[t] = t$

"if" $t = f(\underline{t}_1, \dots, \underline{t}_n)$, $\underline{t}_i \in \underline{T}(\underline{U}, \underline{Z})$,

"then" $\underline{term}[t] = \underline{u} \ [\ \underline{top}[\underline{t}_1] \ [\ \underline{comp}[\underline{t}_1] \] \ ; \ \dots \ ; \ \underline{top}[\underline{t}_n] \ [\ \underline{comp}[\underline{t}_n] \] \]$

"if" $t = e(\underline{t}_1, \dots, \underline{t}_n)$, $\underline{t}_i \in \underline{T}(\underline{U}, \underline{Z})$,

"then" $\underline{term}[t] = e \ [\ \underline{comp}[\underline{t}_1] \ ; \ \dots \ ; \ \underline{comp}[\underline{t}_n] \]$

"if" $t = e(\underline{t}_1, \dots, \underline{t}_n)$, $\underline{t}_i \in \underline{T}(\underline{U}, \underline{Z})$,

"then" $\underline{comp}[t] = \underline{comp}[\underline{t}_1] * \dots * \underline{comp}[\underline{t}_n]$

"if" $t = f \equiv f()$,

"then" $\underline{comp}[t] = t$

"if" $t = f(\underline{t}_1, \dots, \underline{t}_n)$, $\underline{t}_i \in \underline{T}(\underline{U}, \underline{Z})$,

"then" $\underline{comp}[t] = \underline{term}[t]$

In the appendix, III, we first give a major part of the ordered FSTT for RED/2 [Ref. 2], and then we give the simplified version according to the above definition; thus we also give the input grammar, G_i for RED/2.

Discussion 7.9

The idea of ordered sets (i.e. of sequences) of rules, (commands, statements, expressions, etc.) is deeply entrenched in programming.

Our introduction of ordered sets of fstt's and the meaning of the order and the individual fstt's borrows from both Markov's Normal Algorithm's, MNA, [Ref. 11] and from LISP's conditional expression. A few comments may be in order.

MNA's (in their original form) operate on linear strings. The application of a rule in an MNA to a given linear string, transforms that string into another linear string. Rules are applied deterministically, i.e. behind a set of rules constituting an MNA one finds the implicit definition of an evaluator. The evaluator allways starts at the very left end of the string and scan it left-to-right, looking for an applicable substring. Nothing is assumed about the structure of the initial-, the intermediate- nor the final strings. Symbols in any such string denote themselves, i.e. they are constants. The concept of variable is not required, nor commonly introduced.

In LISP the items being transformed are not strings but binary trees. The application of some clause in a conditional expression is, however, based on the concept of an expression (the premise) (a subtree) -- not denoting itself, but -- denoting some value. The evaluation of such an expression, however, does in and by itself require the transformation of a subtree, in this case the reduction of a subtree into a single node whose label is some (of the two) truth value denotations. If the value of the premise is true then some other subtree is evaluated and the result is the value of the whole conditional expression. In appendix I we show how to model

this process of evaluating LISP program doublets as a set of transformations on trees involving nodes and subtrees denoting themselves.

The semantics of LISP given in [Ref. 12] presumes innermost evaluation of expressions, and we shall therefore assume that the LISP FSTT given in appendix I is embedded in the leftmost innermost evaluator Vli.

8. SERIAL TREE MACHINES

We shall now develop an algorithm which applies to programs $\underline{C}$ in a (context-free) tree programming language and yields a finite state tree 'walking' machine of a simple kind. Wherever in the sequel we use the word 'production', one can substitute the word: 'ramification'. Ramifications are defined by the ranking relation on the tree node labelling alphabets.

This section will have two parts. First we informally describe the manner in which the tree machines to be constructed operate on the trees: how and what they read, how they walk on the tree, and what they output. Then we give the algorithm. It has two parts. The first is long and involved. The second is short and easy. The first part has three subparts, with the algorithm alternating cyclically among the three. This first part of the algorithm constructs a finite directed acyclic graph. From this graph the second part reads off the next-state-, the move- and the output functions of the tree machine. The input to the algorithm is the finite ordered set of all fstt's in $\underline{C}$. The output from the algorithm is the finite state tree walking machine, $\underline{M}$, which recognizes the ordered set of all fstt's in $\underline{C}$.

Exactly one machine $\underline{M}$ will be associated with a proper FSTT. It is the job of this $\underline{M}$ to examine a finite part of the tree above the root, which is always labelled with some $f \in \underline{F}$, in order to find

out whether a prefix of that subtree is equal to ('matches' or 'fit') the lt of some $fstt$. We shall not in this section bother to explain who associates the machine M with the node(s) labelled f ; this will be the purpose of section 9 (part II).

There are basically two algorithms for constructing M . Either we construct a machine which starts out looking exclusively for the lt of the first $fstt$ in C . If the interior tree at which M happened to be located did not 'match' or 'fit' that lt , then M would go on looking exclusively for the lt on the next $fstt$ in C . And so forth. For this kind of tree machine the number of states, the moves and everything, would depend linearly on the number of $fstt$'s and on the individual complexity of their lt parts. Such a machine would not be very efficient, in that it would not be capable of making use of prefix trees already recognized and common to the prefix of two or more $fstt$ lt 's. The other algorithm, the one that we are presenting, will indeed make use of all such available information gathered while traversing the interior of the tree under transformation. The resulting machine is then, subject to our constraints, minimal.

8.1 Informal Exposé and some definitions

Let t be a tree subject to transformation, i.e. with at least one node labelled with some $f \in F$, say $\underline{f}$. Suppose that the M machine at some time will be initialized at some such tree node labelled $\underline{f}$. It will then traverse a finite prefix of the subtree rooted in $\underline{f}$.

Depending on what $\underline{M}$ discovers during this traversal, the $\underline{M}_i$ will either 'apply' some f_{stt} in $\underline{C}$, or it will -- in this section -- give up, not applying anything, only cancel its own existence in the tree t .

Let at some time in the traversal process $\underline{M}$ find itself at some node labelled $\underline{u}$ ($\underline{u} \in \underline{U} = \underline{E} + \underline{F}$) and in some state $s \in \underline{S}$, where $\underline{S}$ is the finite state set of $\underline{M}$. Let the ordered successors of $\underline{u}$ be $\underline{u}_1, \underline{u}_2, \dots, \underline{u}_l$. If $l=0$ then $\underline{u}$ is taken to be a leaf node. Let the path from the root, $\underline{top}[t]$, of the tree to $\underline{u}$ be: $\underline{wh}[\underline{ih}] * \dots * \underline{w}_1[\underline{i}_1]$, where $\underline{w}_1 = \underline{top}[t]$, $h+1$ is the level of node $\underline{u}$ and $\underline{wh}$ the direct ancestor of $\underline{u}$ (with $\underline{u}$ being the $\underline{ih}$ 'th left-to-right ordered direct successor of $\underline{wh}$). (We shall choose to write $[\underline{i}(j)]$ instead of $[\underline{ij}]$, namely where j is some simple arithmetic expression). The $[\underline{ij}]$'s are branch selectors.

The machine $\underline{M}$ now reads, then moves, then performs a state transition and may finally output something. These four aspects to the behavior of $\underline{M}$ are now explained:

The $\underline{M}$ reads as it input the ramification (or production):

$$\underline{u} ::= \underline{u}_1 * \underline{u}_2 * \dots * \underline{u}_l$$

Then the $\underline{M}$ machine may move in either of three ways:

$\underline{up}-[k]$: The $\underline{M}$ chooses branch selector $[k]$, one of the l branches up

from $\underline{u}$, and moves up to the corresponding node $\underline{uk}$.

down-k: The $\underline{M}$ chooses to move down k levels, from $\underline{u}$ down to $\underline{w(h+1-k)}$. Such moves always bring the $\underline{M}$ back to the node $\underline{f}$ at which it was originally initialized. Thus k can take any value between 0 and h inclusive.

down-k*UP[p]: A move of the above kind may unconditionally be followed by a move UP (another) path, p , length $[p]$ levels. UP is defined as follows:

"if" $p = v[j]$ "then" $\underline{UP}[p] = \underline{up}[j]$

"if" $p = p' * v[j]$ "then" $\underline{UP}[p] = \underline{UP}[p'] * \underline{up}[j]$

Let $p = p' * v[j]$, where p' may be null, then $p:1 = [j]$. We then have that $p:1 \neq [\underline{i(h+1-k)}]$.

Simultaneously with either of these three moves, the $\underline{M}$ will undergo a state, s , to next-state, s' , transition. The state set $\underline{S}$ has one distinguished initial state s_0 and one distinguished final state s_f , with $s_0 \neq s_f$. Moves of the second type always leads the $\underline{M}$ into the final state.

Only for certain transitions, and these are only some of the transitions into the final state, does the $\underline{M}$ yield an output. It will do so whenever it has recognized a prefix of the tree above it as equal to the $\underline{ltj}$ of some f_{stt} : $[\underline{ltj} =: \underline{rtj}] \in \underline{C}$. The output --

which we shall not explain further -- is the replacement of the subtree, $\underline{s}[\underline{ltj}]$, emanating from the given $\underline{f}$ with the tree $\underline{s}[\underline{rtj}]$.

It should be noted that the above informal description of the behavior of the $\underline{M}$ machine reflects a number of conventions: First we decided to let one machine do all the recognition with respect to $\underline{C}$. Then we decided to let the $\underline{M}$ be present at one and only one node at a time (while in any one state). Then we decided to let the machine be capable of reading as its instantaneous input the ordered sequence of all the direct successors for the node. Finally we decided to group the moves into the three classes given above. Several alternative choices could have been made. The ones we took certainly flavors the algorithm yet to be presented.

The main reason however for the rather unfortunate complexity of this algorithm is due however to our desire of producing the minimal state machine for the recognition of an ordered set of fstt $\underline{lt}$'s.

Based on the above informal description it is now easy to see that the resulting machines all will be finite, that they will be definite in the sense of recognizing only a finite set of 'strings' (which are trees) -- and that they therefore will have no loops, i.e. the state graph for the machines will also be an acyclic directed graph.

8.2 The Algorithm

Recall first that the $\underline{M}$ is to look for an interior satisfying the lt_1 of the first fstt in $\underline{C}$, then recall that if the $\underline{M}$ decides that the present subtree (emanating from $\underline{f}$) cannot satisfy that first lt_1 , then the $\underline{M}$ is not only supposed to go on looking for an interior matching the next lt_2 , etc., but that if in the course of looking for the lt_j of the j 'th fstt it had recognized some prefix of the k 'th fstt lt_k , (k larger than j) then the $\underline{M}$ was to remember that, so that it would not have to re-read that (entire) prefix tree in case the search for lt_j failed.

With each state of $\underline{M}$ we shall therefore associate a finite set of partially traversed trees; and thus the graph to be constructed first will become a representation of all the lt 's together with partial traversals of these. Each lt is finite, and their number is finite, thus there will be a finite number of possible partial traversals of these -- and hence the graph to be constructed will be finite. If we have already visited a part of a tree, then we shall not need to revisit that part again, and thus the graph will be not only directed, but also acyclic.

The graph will have one distinct initial node, p_0 , and one distinct final node, p_f . All nodes of the graph will be partitioned into three disjoint classes: the set of production nodes, the set of move nodes and the set of index nodes. From production nodes we shall derive index nodes, from index nodes we shall derive move nodes, and from move nodes we shall derive either production nodes

-- or the final node. The graph will have one distinguished initial node, which is a production node, and one distinguished final node, which we will call a production node. The links of the graph are directed according to the derivation of the nodes. A link is classified according to the node from which it emanates. A sequence of links, leading from a production node to a production node with no intervening production nodes will come to correspond to a state to next state transition. With such a transition we associated the reading of a production input, a move and a possible output. The links of the graph will come to be labelled with exactly these quantities.

Production links are labelled with productions (or just: ramifications). If $\underline{pn}$ is a production node and $(\underline{in_0}) \underline{in_1}, \dots, \underline{in_q}$ its immediate successor index nodes then at most one of the links from $\underline{pn}$ to (some) $\underline{in_j}$ will receive one or more such production labels. All other links will get exactly one production as its label.

Move links are labelled with exactly one of the above three move denotations: (1) $\underline{up[k]}$, (2) $\underline{down-k}$ or (3) $\underline{down-k*UP[p]}$.

With each node in the graph we associate (1) a path, p , and (2) an ordered finite set of doublets:

$$X = \{ \underline{x_1}, \underline{x_2}, \dots, \underline{x_k} \}$$

The first member of any doublet is a single fstt (e.g. fstt_1 below) in $\underline{C}$. No two doublets in the same set is associated with the same fstt. The second member of the doublet is a finite ordered set of index pair $([p_{j1}, i_s] \text{ below})_s$ (to be defined). Each index pair has a path part (p_{j1} below) and an ordered set of branch selectors ($i_s = [i_{j1}, \dots, i_{jm}] \text{ below}$):

$$X_j = [\text{fstt}_1, [[p_{j1}, [i_{j1}, i_{j2}, \dots, i_{jm}]], \dots]]$$

We now state an observation. It can only be verified once the algorithm has been described in full, but the early mentioning should facilitate the understanding of the algorithm. It runs as follows: With each graph node we can associate a unique node of some unique lt part of an fstt:

- (1) Take the first doublet in the set attached to the node. The path part of the first member of the index set will be so derived as to point to a unique node of the lt for the fstt of that doublet. This is the tree node uniquely identified. Speaking in terms of the $\underline{M}$ machine we can say that it will be located at this unique node while in the state associated with the graph node.
- (2) If one or more of the subsequent doublets have paths coinciding with the one mentioned above, then all corresponding lt trees will have that partial path in common. We say that all these

lt 's are presently being considered by the M while in the state denoted by the graph node.

- (3) The $fstt$'s of all remaining doublets define a set of subtrees: $s[lt]$. For some of these, the path mentioned above will be a partial path from f , for other subtrees there will be no corresponding partial paths.

We now present the three sub-constructions: (1) the production partitioning, (2) the index set update and (3) the path update. Following this we give the description of both parts of the algorithm. A formal proof that the algorithm indeed computes what we intend to be computed is not presented in this (part I of the) report.

We need a few extra definitions:

Definition 8.1.1.1 select $[t,p]$

Let $t \in T(\underline{U}, \underline{Z})$ and let p be some path. The construction now to be defined attempts to find a path of t for which p is a prefix, and if successful it returns the subtree 'selected' by p in t ; the definition states what is meant by selecting such a subtree.

"if" $p = \underline{null}$, "then" select $[t,p] = t$

"if" $t \in \underline{U+Z}$ & $p \neq \underline{null}$, "then" select $[t,p] = ?$

"if" $t = u(\underline{t}_1, \dots, \underline{t}_n) \ \& \ p = v[i]*p'$

"then"

"if" $u=v \ \& \ 1[i]n$, "then" $\underline{select}[t,p] = \underline{select}[\underline{t}_i, p']$

"else" $\underline{select}[t,p] = ?$

Definition 8.1.1.2 $\underline{prod}[t,p] = \underline{ram}[\underline{select}[t,p]]$

Definition 8.1.1.3 $\underline{ram}[t]$

Taking $\underline{ram}$ of a tree means to return the root of the tree and its ordered sequence of direct descendants (if any):

"if" $t \in \underline{U+Z}$, "then" $\underline{ram}[t] = t$

"if" $t = u(\underline{t}_1, \dots, \underline{t}_n)$,

"then" $\underline{ram}[t] = \underline{top}[t](\underline{top}[\underline{t}_1]*\dots*\underline{top}[\underline{t}_n])$

Definition 8.1.1.4 $\underline{index}[t,p]$

Taking the $\underline{index}$ of a tree t with respect to a (partial) path p results in an ordered sequence of branch selectors as found on the node selected by p in t :

$\underline{index}[t,p] =$

if $t=\underline{null}$ then $\emptyset$

else

```

if p=null then
    if t = w ∈ U+Z then  $\emptyset$ 
        else
            if t = u(t1, ..., tn)
                then [top[t1][1], ..., top[tm][m]]
                    else
let t = u(t1, ..., tn) and p = p'*v[j]
if u=v and 1[j]n
    then index[tj, p']
    else  $\emptyset$ 

```

8.2.1 Production Partitioning

Let p_n denote a production node; let $X = \{X_1, \dots, X_k\}$ be the ordered set of doublets associated with p_n ; let p be the path label associated with p_n and let $X_j \in X$. Let X_{j1} denote the left-part of the fstt associated with X_j .

Let the number of all possible productions of G (or ramifications found in trees) be q and number all these, R_i , distinctly 1 through q .

The production (or ramification) partitioning is a partitioning of the set X into at most $q+1$ sets, $(Y_0), Y_1, \dots, Y_q$. Some or all of these may overlap; for i larger than 0:

$$Y_i = \{ X_j \mid \text{prod}[X_{j1}, p] = R_i \text{ or } \text{prod}[X_{j1}, p] = ? \}$$

$$Y_0 = \text{if there is at least one } Y_i = \emptyset \\ \text{then } \{ X_j \mid \text{prod}[X_{j1}, p] = ? \}$$

The set Y_0 is only computed if in computing $Y_1, \dots, Y_q$, not all productions in G were found in computing some $\text{prod}[X_{j1}, p]$. The set Y_0 may, according to this definition, be the null set. All sets, including Y_0 if computed, are ordered in doublets according to the order of these in X . Let r be the number of non-null sets Y_i , then p_n gets r successor index nodes: $(i_0), i_1, i_2, \dots, i_s$ (where, if Y_0 is computed, $s=r$, else $s=r-1$). Let i_j be the index node associated with the set Y_i , derived from p_n , then the link from p_n to (each such) i_j will be distinctly labelled with (the production or ramification) R_i .

If Y_0 is computed, then two cases arise: either $Y_0 = \emptyset$ or $Y_0 \neq \emptyset$. In the first case i_0 is specially marked, and the links leading from i_0 to the final node can be constructed immediately. In the latter case, no special provisions need be made, i_0 can be treated on par with any other i_j .

The link leading from p_n to a specially marked i_0 will be labelled with all those productions in G not already the label of some link from p_n . The i_0 node will be followed by a move node, m_0 , and the link from m_0 to the final node will be labelled with $\text{down-length}[p]$.

(Transitions of this kind correspond to the evaluator not finding an applicable interior rooted in $\underline{f}$. As indicated, e.g. in $\underline{V}_n$, the evaluator (shall) then make the decision to go on according to rule 13:)).

The initial node has the null path label and all the fstt's of $\underline{C}$ associated with it. Otherwise the content of all doublets is the empty set $\emptyset$. Therefore $\text{prod}[\underline{X}_{j1}, \underline{\text{null}}]$ is always defined and $\underline{Y}_0$ if computed is therefore also $\emptyset$. If computed the link from $\underline{p}_1$ to $\underline{i}_{10}$ is constructed, and the link label from $\underline{m}_{10}$ to $\underline{p}_f$ will thus become: down-0 !

8.2.2 Index Set Update

Let $\underline{i}_n$ be an index node; $\underline{p}$ be the corresponding label; let the ordered set, $\underline{X}$, of doublets be as above and let $\underline{I}_j$ be the ordered set of index pairs associated with $\underline{X}_j$. That is:

$$\underline{I}_j = \{[\underline{p}_1, \underline{\text{index}}[\underline{X}_{j1}, \underline{p}_1]], \dots, [\underline{p}_p, \underline{\text{index}}[\underline{X}_{j1}, \underline{p}_p]]\}$$

Then for each such $\underline{X}_j \in \underline{X}$ update $\underline{I}_j$:

$$\underline{I}_j' = \{[\underline{p}, \underline{\text{index}}[\underline{X}_{j1}, \underline{p}]], \underline{I}_j\}$$

preserving the order of $\underline{I}_j$. If $[\underline{p}, \underline{\text{index}}[\underline{X}_{j1}, \underline{p}]] = \emptyset$ then $\underline{I}_j' = \underline{I}_j$.

To each index node in there then corresponds exactly one move node, mn, with the same path label, p, and with the ordered doublet set:

$$X' = \{X_1', \dots, X_k'\}$$

such that:

$$\text{if } X_j = [fstt, I_j] \in X, \text{ then } X_j' = [fstt, I_j']$$

8.2.3 Path Construction

Let mn denote a move node with path label p and with doublet set X as above. Let:

$$X_1 = [fstt, [p_{11}, [\underline{up}[j_1], \dots, \underline{up}[j_l]], \dots]]$$

where l is larger than 0. Let pn be the single production node to be derived from mn. We are now to label the move link from mn to pn with some (one distinct) move from one of the three classes:

- (1) If $p=p_{11}$ then the move to be made is $\underline{up}[j_1]$. pn is then to receive the updated path label: $u[j_1]*p$, where u is the label of the corresponding node in the left-part tree of the fstt. The ordered set X' of doublets of pn is derived from X in the following way: If in some X_j there is an index member set: $[p_{11}, I]$ then in X_j' only this member may need a change. This member is updated by deleting $\underline{up}[j_1]$ from I -- if it is a member of I, otherwise no update is required. We write this:

$[p11, I/\underline{up}[j1]]$. The doublet $fstt$ and all other members of the index sets for which no (such) combination of $p11$ and $\underline{up}[j1]$ occur are carried over unchanged, into X_j' , preserving the order given in X_j .

- (2) If there is at least one member of the index set of X_1 , and if for no such member we have $p=p11$, then the labelling of $\underline{pn}$ and the move link to $\underline{pn}$ is derived as follows: Let the first member of the index set be as above. Let $p=p2*p1$, $p11=p3*p1$; and let the last item of $p2$, respectively $p3$, be different: i.e. $p2:1 \neq p3:1$. Then the move has two parts: first a move down-length $[p2]$, followed by a move up $\underline{UP}[j1*p3]$. The move label is accordingly. The path label of $\underline{pn}$ becomes: $[j1]*p11$. The set X' of doublets associated with $\underline{pn}$ is derived from X in a manner very much like before: where we then used p ($=p11$) and $[j1]$ for the mapping of X into X' we now use $p11$ and $[j1]$ instead.

- (3) If there are no index members of X_1 , then $\underline{pn}$ is the final (production) node. We have recognized the left-part tree of the corresponding $fstt$. $\underline{pn}$ receives the path label null (as does $\underline{p1}$). The move label is down-length $[p]$. If k is largerer than 1 then the doublets $X_2, \dots, X_k$ are deleted, together with X_1 , is deleted at $\underline{pf}$. They ($X_2, \dots$) are said to be overridden by X_1 .

8.2.5 Machine Definition

The formalization of the tree walking machine now to be constructed amounts to a 9-tuple (sorry for that!):

$$\underline{M} = [\underline{S}, \underline{P}, s_0, sf, \underline{f}, \underline{g}, \underline{h}, \underline{C}, \underline{T}(\underline{E}+\underline{F})]$$

$\underline{S}$ and $\underline{P}$ is the finite sets of respectively states and productions (ramifications). The initial state is s_0 , and the final state is sf . $\underline{f}$, $\underline{g}$ and $\underline{h}$ are the next-state-, the move- and the output functions, respectively. $\underline{C}$ is the finite ordered set of fstt's associated with $\underline{M}$ and $\underline{T}(\underline{E}+\underline{F})$ is a set of complete trees. $\underline{M}$ is initially started up at the root ($\underline{f}$) of some (complete) suffix tree, $t \in \underline{T}(\underline{E}+\underline{F})$. The three functions satisfy the following expressions, where $s_i, s_j \in \underline{S}$, $p_k (= p_{k1} \text{ and } := p_{kr}) \in \underline{P}$: where $l_{tm} := r_{tm} \in \underline{C}$:

$$\underline{f}[s_i, \text{null}, t] = s_i$$

$$\underline{f}[s_i, p_k, t] = s_j$$

$$\underline{g}[s_i, \text{null}, t] = \underline{\text{down}}-0$$

$$\underline{g}[s_i, p_k, t] =$$

$$\underline{\text{either:}} \quad \underline{\text{up}}[1], 1 \quad [\quad 1 \quad [\quad \underline{\text{length}}[p_{kr}]$$

$$\underline{\text{or:}} \quad \underline{\text{down}}-1, 0 \quad [\quad 1 \quad [\quad \text{MAX}-m(\underline{\text{height}}[l_{tm}])$$

$$\underline{\text{or:}} \quad \underline{\text{UP}}[p] * \underline{\text{down}}-1, 1 \text{ as above, and:}$$

where p is a partial path of some path in some l_{tm} .

$$\begin{aligned}
 \underline{h}[\underline{si}, \underline{null}, t] &= t \\
 \underline{h}[\underline{si}, \underline{pk}, t] &= (\underline{if} \ \underline{f}[\underline{si}, \underline{pk}, t] = \underline{sf} \\
 &\quad \underline{then} \ \underline{either} \ \underline{s}[\underline{rtm}] \\
 &\quad \quad \underline{or} \ t \\
 &\quad \underline{else} \ t)
 \end{aligned}$$

8.2.6 Machine Construction

With the acyclic graph we now associate a distinct state s with distinct prefix nodes. The initial (production) node, p_0 , is thus relabeled into s_0 and the final production node, p_f into s_f . This labelling results in $\underline{S}$.

Let si be the state associated with some prefix node other than p_f . Let the direct successor states of si be $sj_1, sj_2, \dots, sj_n$. Let the production set labelling on the link from ' si ' towards ' sj_1 ' ($1 \leq j \leq n$) be $\underline{p}_{j1}$. Let the move label on the corresponding move link be $\underline{mj}_1$, then:

all $\underline{j}_1$, all $\underline{p}_{j1} \in \underline{p}_{j1}$,

$\underline{f}[\underline{si}, \underline{p}_{j1}, t] = \underline{sj}_1$

$\underline{g}[\underline{si}, \underline{p}_{j1}, t] = \underline{mj}_1$

$\underline{h}[\underline{si}, \underline{p}_{j1}, t] =$

if $\underline{sj}_1 \neq \underline{sf}$

then t

else

comment: let $\underline{x}_1$ be the first doublet of the set associated

with the index- or move node 'between' si and sjl , and let the $fstt$ associated with this doublet be: $[l_{tm}=:r_{tm}]$, then: end

$s[r_{tm}]$

comment: where s is the substitution operation defined (through a) by t and the applicable $fstt$: $[l_{tm}=:r_{tm}]$ end;

For all other, so far unspecified entries, i.e. for all $p \notin \text{set-union}(jl) \ Pjl$:

$f[si,p,t]$ = sf

$g[si,p,t]$ = down-length $[pth]$

where pth is the path label
associated with the si node.

$h[si,p,t]$ = t

This concludes the description of the algorithm steps. The complete algorithm then starts out by associating all of C with p_0 and the path label null with that node. The algorithm second part again starts at p_0 and stops when all state to next-state sequences in the acyclic graph has been read off.

If in the process of constructing the acyclic graph two or more move nodes have identical path labels and identical ordered sets of doublets then we can merge all such move nodes into one. The resulting machine will be minimal state under the assumptions made with respect to possible inputs and moves.

Example

Using only the rho and delta part of the RED/2 FSTT of appendix III, appendix V shows the resulting serial tree machine. Note that to each operator one can associate a distinct, parallel decomposable, serial tree machine; thus there are as many such tree machines as there are distinct operators in F to any given FSTT.

(Notation: an 'up arrow' and a digit, k , denotes up[k], whereas a 'down arrow' and a digit, k , denotes down[k]. A number, q , within parentheses, (q), names the like-numbered fstt within the ordered set of fstt's).

REMARK:

If one was to move fstt (12) up as the first fstt then the whole delta part would collapse to just one single tree machine transition; by inserting (10) a few, 2 or 3, fstt rules up, the behavior of the tree machine would again be drastically changed -- reflecting the altered semantics of -- in this case the RED/2 -- FSTT.

CONCLUSION

This concludes part I. In part II we shall start out by making appropriate remarks concerning the nature of the universal evaluator-evaluator, VV, whose job it is to schedule the distribution of the constructed serial tree machine(s) over the appropriate trees. A number of other questions, such as e.g. the relation to variable-free programming [Refs. 4 & 2], will also be investigated there. Finally we shall analyse the power of the notions extensively defined and some restrictions thereof. One interpreter for RED/2 is implemented on the basis of serial tree machines. Together with an evaluation scheme (e.g. V_{li} or V_n) this implementation amounts to a single push-down tree machine. Needless to say: we would of course suggest similar, mechanizable, implementations for LISP- and AE-like languages, yes even parts of the semantics for ALGOL-like (imperative) languages. It is a special property of primary languages [Ref. 2] of which RED/2 is an example that such implementation then can be expressed in terms of a push-down machine traversing trees in T(E) only, with all operators mapped onto the stack; this subject will be further covered in part II.

ACKNOWLEDGEMENT

I am indebted to John Backus for having provided the RED/2 language. His work helped clarify the notion of finite state tree computations and enabled me to show the definitional relation between the LISP-like languages (with variables and environment) and the RED-like languages (requiring no environment due to the variable-freeness). His constant vigilancy, and repeated, but maybe not always successful, calls for clarity, economy and simplicity of concepts, is deeply appreciated. I finally thank him for having provided also the free environment in which this investigation was carried out.

REFERENCES

- [1] Aho, A.V., and Ullman, J.D.: 'Translations on a Context Free Grammar', Information and Control, vol. 19, no. 5, December 1971, pp. 439-475.
 - [2] Backus, J.: 'Reduction Languages and Variable-free programming', IBM Research report, RJ 1010, San Jose, April 7, 1972, 112 pp.
 - [3] Chomsky, N: "Aspects of the theory of Syntax", MIT press, Cambridge, Mass. 1965, 250 pp.
 - [4] Curry, H.B. and Feys, R.: "Combinatory Logic, vol. I", North-Holland Publ., Amsterdam, 1968, 417 pp.
 - [5] Knuth, D.E.: 'Semantics of Context Free Languages', Mathematical Systems Theory, vol. 2, 1968, pp. 127-146 (for errata, see: 'ibid' vol. 5, 1971, pp. 95-96).
- and:
- 'Examples of Formal Semantics', in: (Lecture Notes in Mathematics: vol. 188), "Symposium on Semantics of Algorithmic Languages", Springer-Verlag, 1971, pp. 212-235.
 - [6] Landin, P.J.: 'The Mechanical Evaluation of Expressions', Computer Journal, vol. 6, 1964, pp. 308-320.
 - [7] Landin, P.J.: 'A correspondence between ALGOL 60 and Church's Lambda notation', Communications of the A.C.M., vol. 8, nos. 2-3, 1965, pp. 89-101, pp. 158-165.
 - [8] Lewis, P.M. III. and Stearns, R.E.: 'Syntax Directed Transductions', Journal of the A.C.M., vol. 15, no. 2, 1968, pp. 465-488.
 - [9] Lucas, P. and Walk, K.: 'On the Formal Description of PL/1', in: "Annual Review in Automatic Programming", vol. 6, part 3, 1969, Pergamon Press, pp. 105-182.

- [10] Manna, Z., Ness, S. and Vuillemin, J.: 'Inductive methods for proving properties of programs', in: "Proceedings of (an) ACM conference on Proving Assertions about Programs", ACM, New York, January 1972.
- [11] Markov, A.A.: "The Theory of Algorithms", (translated from the Russian:) U.S. Dept. of Commerce, Office of Technical Services, report no.: OTS 60-51085.
- [12] McCarthy, J. et.al.: "LISP 1.5 Programmers Manual", MIT Press, Cambridge, Mass. 1965, pp. 106.
- [13] Morris, F. Lockwood: 'The Next 700 Programming Language Semantics', unpublished notes, University of Essex, England, 1970.
- [14] Rounds, W.C.: 'Mappings and Grammars on Trees', Mathematical Systems Theory, vol. 4, no. 3, 1970, pp. 257-287.
- [15] Reynolds, John C.: 1972 ACM Conference, Boston, August 1972.
- [16] Thatcher, J.W.: 'Characterizing Derivation Trees of Context Free Grammars through a generalization of Finite Automata Theory', Journal of Computer and Systems Sciences, vol. 1, 1969, pp. 317-322.
- [17] Thatcher, J.W. and Wright, J.B.: 'Generalized Finite Automata Theory with an Application to a Decision Problem of 2nd order Logic', Mathematical Systems Theory, vol. 2, pp. 57-81, 1968.
- [18] Thatcher, J.W.: 'Generalized**2 Sequential Machine Maps', 'ibid', vol. 4, 1970, pp. 339-367.
- [19] Wirth, N. and Weber, H.: 'EULER - A Generalization of ALGOL, and its formal definition', Communications of the A.C.M., vol. 9, nos 1-2, 1966, pp. 13-25, pp. 89-99.
- [20] Zemanek, H.: 'Was ist Informatik?', Elektronische Rechenanlagen, vol. 13, no. 4, 1971, pp. 157-161.

APPENDICES I II III IV V

APPENDIX I: The pure LISP 1.5 FSTT:

```

(0) evalquote[S1,S2]                =: apply[S1,S2,S(A(NIL))]

      apply[S(A(****)),S1,S2]
(1)      **** = CAR                  =: car[car[S1]]
(2)      **** = CDR                  =: car[cdr[S1]]
(3)      **** = CONS                 =: S(car[S1],car[cdr[S1]])
(4)      **** = ATOM                 =: atom[car[S1]]
(5)      **** = EQ                   =: eq[car[S1],car[cdr[S1]]]
(6)      **** = V1                   =: apply[eval[S(A(V1))],S3,S2,S3]

(7) apply[S(S(A(LAMBDA))),S1,S2,S3] =: eval[car[cdr[S1]],pairlis[car[S1],S2,S3]]
(8) apply[S(S(A(LABEL))),S1,S2,S3]  =: apply[car[cdr[S1]],S2,S(S(car[S1],car[cdr[S1]]),S3)]

(9) eval[S(A1),S2]                  =: cdr[assoc[A1,S2]]
(10) eval[S(S(A(QUOTE))),S1,S3]     =: car[S1]
(11) eval[S(S(A(COND))),S1,S2]      =: evcon[S1,S2]
(12) eval[S(S(A(V1))),S2,S3]        =: apply[S(A(V1)),evlis[S2,S3],S3]
(13) eval[S(S(S1,S2),S3),S4]        =: apply[S(S1,S2),evlis[S3,S4],S4]

(14) evcon[S(A1),S2]                =: S(A(?))
(15) evcon[S(S1,S2),S3]             =: tevcon[eval[car[S1],S2],S(S1,S2),S2]

```

- | | | | |
|------|---|----|---|
| (16) | tevcon[S(A(TRUE)), <u>S1</u> , <u>S2</u>] | =: | eval[car[cdr[car[<u>S1</u>]]], <u>S2</u>] |
| (17) | tevcon[S(A(FALSE)), <u>S1</u> , <u>S2</u>] | =: | evcon[cdr[<u>S1</u>], <u>S2</u>] |
| (18) | evlis[S(A(NIL)), <u>S1</u>] | =: | S(A(NIL)) |
| (19) | evlis[S(<u>S1</u> , <u>S2</u>), <u>S3</u>] | =: | S(eval[<u>S1</u> , <u>S3</u>], evlis[<u>S2</u> , <u>S3</u>]) |
| (20) | pairlis[S(A(NIL)), <u>S1</u> , <u>S2</u>] | =: | <u>S2</u> |
| (21) | pairlis[S(<u>S1</u> , <u>S2</u>), S(<u>S3</u> , <u>S4</u>), <u>S5</u>] | =: | S(S(<u>S1</u> , <u>S3</u>), pairlis[<u>S2</u> , <u>S4</u> , <u>S5</u>]) |
| (22) | assoc[<u>S1</u> , <u>S2</u>] | =: | tassoc[equal[car[car[<u>S2</u>]], <u>S1</u>], <u>S1</u> , <u>S2</u>] |
| ZTT# | tassoc[S(A(TRUE)), <u>S1</u> , <u>S2</u>] | =: | car[<u>S2</u>] |
| (24) | tassoc[S(A(FALSE)), <u>S1</u> , <u>S2</u>] | =: | assoc[<u>S1</u> , cdr[<u>S2</u>]] |
| (25) | equal[S(A(<u>id1</u>)), S(A(<u>id2</u>))] | =: | eq[<u>id1</u> , <u>id2</u>] |
| (26) | equal[S(<u>A1</u>), S(<u>S1</u> , <u>S2</u>)] | =: | S(A(FALSE)) |
| (27) | equal[S(<u>S1</u> , <u>S2</u>), S(<u>A1</u>)] | =: | S(A(FALSE)) |
| (28) | equal[S(<u>S1</u> , <u>S2</u>), S(<u>S3</u> , <u>S4</u>)] | =: | tequal[equal[<u>S1</u> , <u>S3</u>], equal[<u>S2</u> , <u>S4</u>]] |
| (29) | tequal[S(A(TRUE)), S(A(TRUE))] | =: | S(A(TRUE)) |
| (30) | tequal[S(A(FALSE)), S(A(TRUE))] | =: | S(A(FALSE)) |
| (31) | tequal[S(A(TRUE)), S(A(FALSE))] | =: | S(A(FALSE)) |
| (32) | tequal[S(A(FALSE)), S(A(FALSE))] | =: | S(A(FALSE)) |
| (33) | tequal[S(A(FALSE)), [tequal, 1]] | =: | S(A(FALSE)) |
| (34) | tequal[S(A(FALSE)), [equal, 1]] | =: | S(A(FALSE)) |

etcetera ...

```
(35) car[S(S1,S2)]      =: S1

(36) cdr[S(S1,S2)]      =: S2

(37) atom[S(A1)]         =: S(A(TRUE))
(38) atom[S(S1,S2)]      =: S(A(FALSE))

(39) eq[id1,id2]
    id1=A, id2=A          =: S(A(TRUE))
    id1=A, id2=B          =: S(A(FALSE))
    ... ..               =: ...
    id1=Z, id2=Y          =: S(A(FALSE))
    id1=Z, id2=Z          =: S(A(TRUE))
```

APPENDIX II: The Landin 'S.E.C.D' AE FSTT:

The Landin AE SECD FSTT:

- (0) evalquote[A1]
 $\text{=: transform}[S(NIL), E(NIL), C(A1, NIL), D(NIL)]$

- (1) transform[S1, E1, C(I1, C1), D1]
 $\text{=: transform}[S(\text{location}[I1, E1], S1), E1, C1, D1]$

- (2) transform[S1, E1, C(lambda(IL1, B1), C1), D1]
 $\text{=: transform}[S(\text{constructclosure}[\text{environmentpart}[E1, IL1], \text{controlpart}[B1]], S1), E1, C1, D1]$

- (3) transform[S1, E1, C(combination(A1, A2), C1), D1]
 $\text{=: transform}[S1, E1, C(A2, C(A1, C(\text{app}, C1))) , D1]$

- (4) transform[S(constructclosure(environmentpart[E1, IL1], controlpart[B1]), S(S1, S2)), E2, C(app, C1), D1]
 $\text{=: transform}[S(NIL), \text{derive-assoc}[IL1, S1, E1], B1, D(S2, E2, C1, D1)]$

- (5) transform[S(S1, S(S2, S3)), E1, C(app, C1), D1]
 $\text{=: transform}[S(\text{application}[S1, S2], S3), E1, C1, D1]$

- (6) transform[S(S1, S2), E1, NIL, D(S3, E2, C1, D1)]
 $\text{=: transform}[S(S1, S3), E2, C1, D1]$

1) location[I1,NIL] =: ?
2) location[I1,E(i-v-pair(I2,V1),E1)] =: tlocation[eq[I1,I2],I1,V1,E1]
3) tlocation[TRUE,I1,V1,E1] =: V1
4) tlocation[FALSE,I1,V1,E1] =: location[I1,E1]
5) derive-assoc[NIL,NIL,E1] =: E1
6) derive-assoc[NIL,S1,E1] =: ?
7) derive-assoc[IL1,NIL,E1] =: ?
8) derive-assoc[I1,S1,E1] =: E(i-v-pair(I1,S1),E1)
9) derive-assoc[IL(I1,IL1),S(V1,S1),E1] =: E(i-v-pair(I1,V1),derive-assoc[IL1,S1,E1])

APPENDIX III: The Backus RED/2 FSTT:

A Concrete Syntax for RED/2:

$I ::= ? \ I \ * \ I \ R \ I \ M \ I \ E \ I \ P \ (\ I \ L \) \ I \ [\ I \ B \ I \]$
 $L ::= I \ L \ I \ \underline{\text{null}}$
 $P ::= * \ I \ R \ I \ M \ I \ E \ I \ \underline{\text{null}}$
 $E = \{ x \mid I \ C ::= * \ x \ \underline{\text{and}} \ x \notin R \cup M \}$
 $R = \text{" set of defined regular function names "}$
 $M = \text{" set of defined meta function names "}$

where: $C ::= C D \mid D$, and D is a set of characters, usually including upper- and lower case letters, digits, and e.g. the following special characters: $\emptyset$, θ , $+$, $-$, $_{-}$, $\&$, $'$, $"$, $/$ and backspace (the latter enabling overprinted characters, such as the first two special characters). Names of regular-, R , and meta-, M , functions are strings composed likewise. E , R and M denote mutually disjoint sets of finite length strings of such characters.

$B ::= \text{'blank-space'} \mid I \mid . \mid I \mid , \mid I \mid : \mid I \mid ; \mid I \mid =$

The Abstract RED/2 Syntax:

$$I ::= ? \mid I * I \mid R \mid M \mid E \mid I \mid P \mid L \mid I \mid I$$

$$L ::= I.L \mid I \mid \underline{\text{null}}$$

P, E, R, M -- as above.

On the basis of the abstract syntax and:

$$\underline{E} = \{ I, *, ?, R, M, E, L, P \}$$

$$\underline{F} = \{ \text{rho}, \text{delta}, \text{psi}, \text{epsilon}, \text{def} \}$$

we now present the more crucial parts of the RED/2 semantics definition as an ordered set, $\underline{C}$, of fstts:

$$(1) \text{rho}[I(\underline{P}, \underline{I}, \underline{L})] =: I(P, \text{rho}[\underline{I}], \text{rho}[\underline{L}])$$

$$(2) \text{rho}[I(I_1, I_2)] =: \text{rho}[I(\text{delta}[\text{rho}[I_1], \text{rho}[I_2]])]$$

$$(3) \text{rho}[\underline{I}] =: \underline{I}$$

$$(4) \text{rho}[L(\underline{I}, \underline{L})] =: L(\text{rho}[\underline{I}], \text{rho}[\underline{L}])$$

$$(5) \text{rho}[L()] =: L()$$

(6) $\text{delta}[I(I(P(*), I_1, L_1), I_2)]$
 $=:$ $\text{delta}[I(I(P(), I_1, L_1), I(P(), I(P(*), I_1, L_1), L(I_2, L()))))]$

(7) $\text{delta}[I(I(M), I)]$
 $=:$ $\text{delta}[I(\text{psi}[I(M)], I(P(), I(M), L(I, L())))]$

(8) $\text{delta}[I(I(P(), I(P(*), I_1, L_1), L_2), I_2)]$
 $=:$ $\text{delta}[I(I(P(), I_1, L_1),$
 $I(P(), I(P(), I(P(*), I_1, L_1), L_2), L(I_2, L())))]$

(9) $\text{delta}[I(I(P(), I(M), L), I)]$
 $=:$ $\text{delta}[I(\text{psi}[I(M)], I(P(), I(P(), I(M), L), L(I, L())))]$

(10) $\text{delta}[I(I(P(), I_1, L()), I_2)]$
 $=:$ $\text{delta}[I(I_1, I_2)]$

(11) $\text{delta}[I(I(P(), I_1, L(I_2, L_1)), I_3)]$
 $=:$ $\text{delta}[I(I_1, I(\text{delta}[I(I(P(), I_2, L_1), I_3)]))]$

(12) $\text{delta}[I(I_1, I_2)]$
 $=:$ $\text{epsilon}[I(I_1, I_2)]$

$\text{psi}[I(M(\text{adj}))]$
 $=:$ $I(P(), I(\text{car}), L(I(\text{rot}), L()))$

$\text{psi}[I(M(\text{condarg}))]$

```

=: I(P(),I(app),L(I(cond),L(I(apply),
    L(I(P(),I(sec),L(I(distfunc),L()))),
    L(I(distfunc),L()))))

```

etcetera

```

epsilon[I(I(R),I)]

```

```

=: delta[I(def[I(R)],I)]

```

```

epsilon[I(I(rev),I(P,I1,L(I2,L())))]

```

```

=: I(P,I2,L(I1,L()))

```

```

epsilon[I(I(cond),I(P,I(t),L(I(P,I1,L(I2,L()))),L())))]

```

```

=: I1

```

```

epsilon[I(I(cond),I(P,I(f),L(I(P,I1,L(I2,L()))),L())))]

```

```

=: I2

```

etcetera

```

def[I(R(t1))]

```

```

=: I(P(),I(car),L(I(sep),L()))

```

```

def[I(R(rot))]

```

$=: I(P(), I(\text{assoc}), L(I(\text{rev}), L(I(\text{assoc}), L(I(\text{rev}), L()))))$

etcetera

Based on the concrete syntax and using term-tree fstt's we can present a somewhat more palatable context-free fstt program $\underline{C'}$:

$\text{rho } [I [P (I L)]]$

$=: I [P (\text{rho } [I] \text{ rho } [L])]$

$\text{rho } [I [[I_1 B I_2]]]$

$=: \text{rho } [I [\text{delta } [I [[\text{rho } [I_1] B \text{ rho } [I_2]]]]]]$

$\text{rho } [I]$

$=: I$

$\text{rho } [L [I L]]$

$=: L [\text{rho } [I] \text{ rho } [L]]$

$\text{rho } [L []]$

$=: L []$

$$\begin{aligned} \text{delta } [I [[*(I_1 L_1) B I_2]]] \\ =: \text{delta } [I [[(I_1 L_1) B (*(I_1 L_1) I_2)]]] \end{aligned}$$

$$\begin{aligned} \text{delta } [I [[M B I]]] \\ =: \text{delta } [I [[\text{psi } [M] B (M B I)]]] \end{aligned}$$

$$\begin{aligned} \text{delta } [I [[(*(I_1 L_1) L_2) B I_2]]] \\ =: \text{delta } [I [[(I_1 L_1) B ((*(I_1 L_1) L_2) I_2)]]] \end{aligned}$$

$$\begin{aligned} \text{delta } [I [[(M L) B I]]] \\ =: \text{delta } [I [[\text{psi } [M] B ((M L) B I)]]] \end{aligned}$$

$$\begin{aligned} \text{delta } [I [[(I_1) B I_2]]] \\ =: \text{delta } [I [[I_1 B I_2]]] \end{aligned}$$

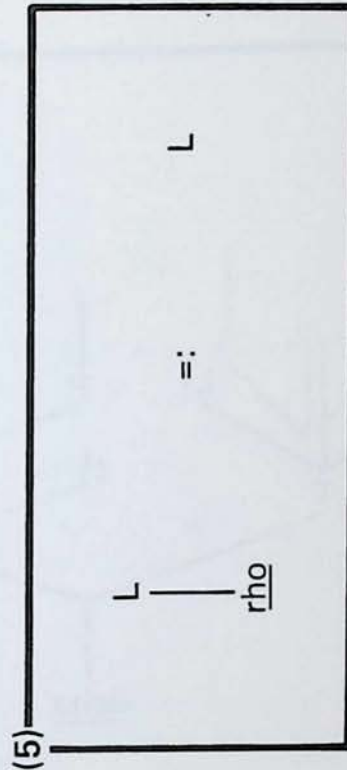
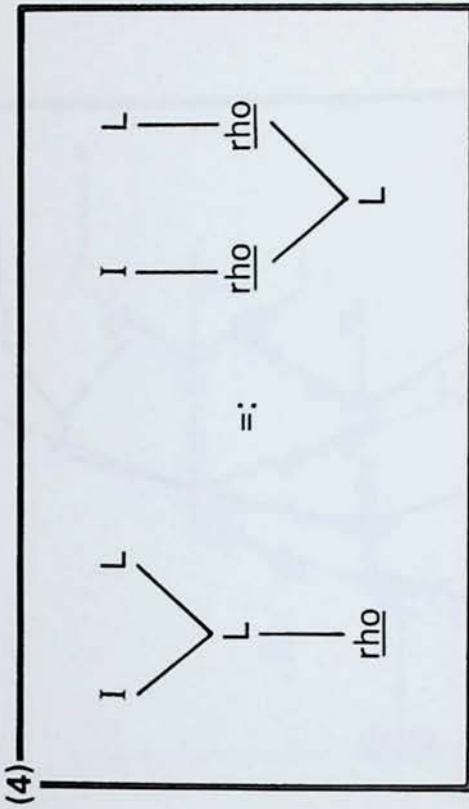
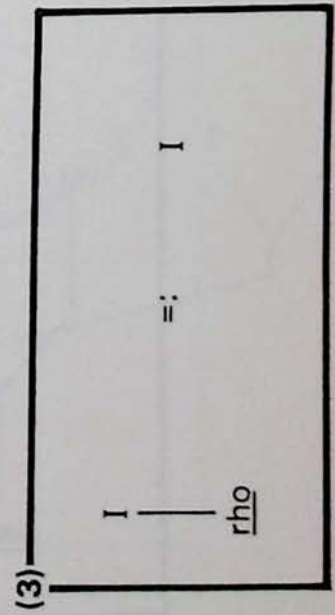
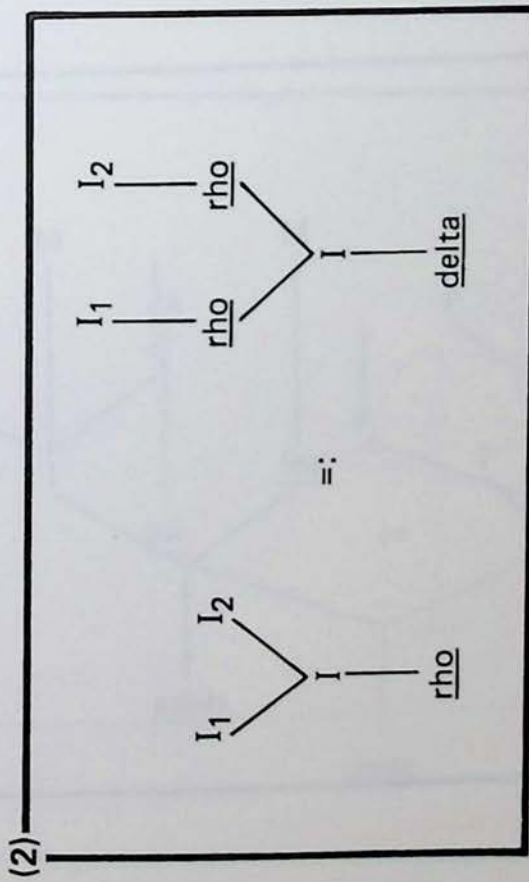
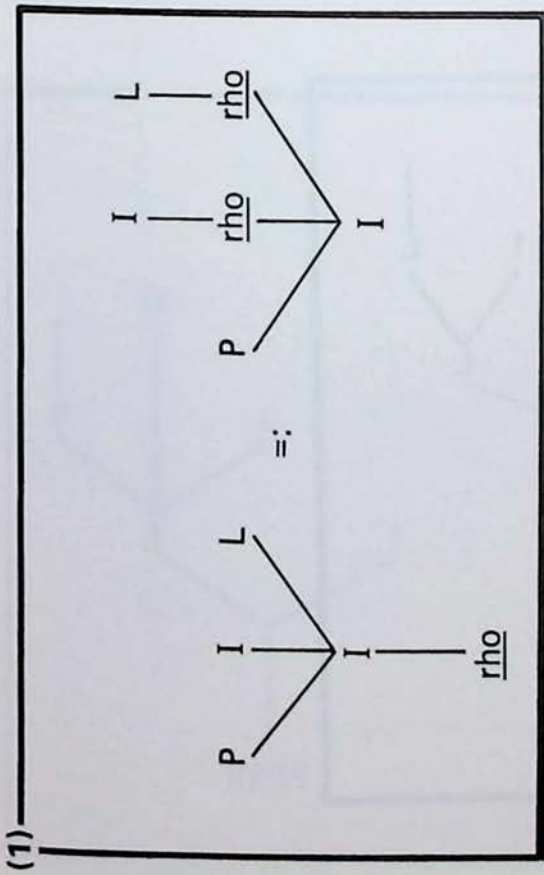
$$\begin{aligned} \text{delta } [I [[[(I_1 B_1 I_2 L_1) B_2 I_3]]]] \\ =: \text{delta } [I [[I_1 B_2 \text{delta } [I [[(I_2 L_1) B_2 I_3]]]]]] \end{aligned}$$

$$\begin{aligned} \text{delta } [I [[I_1 B I_2]]] \\ =: \text{epsilon } [I [[I_1 B I_2]]] \end{aligned}$$

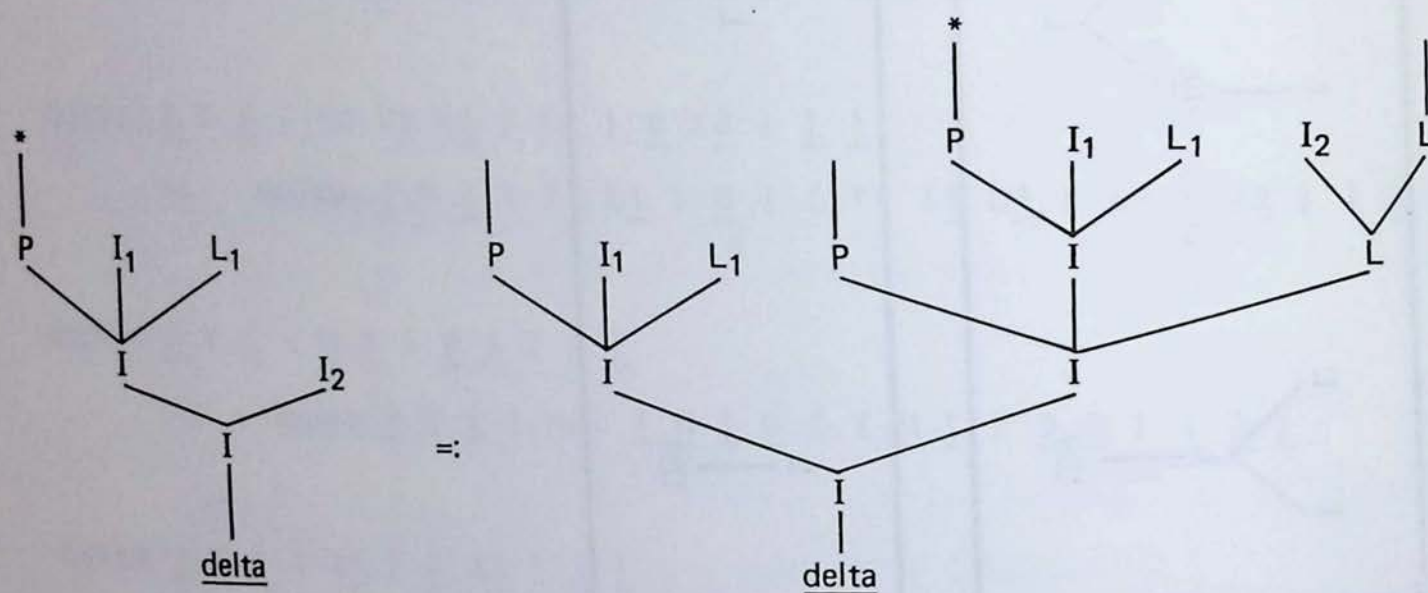
As an example of the real benefit one gets from using term tree fstt's, just give the one example of 'psi' of 'condarg':

$$\begin{aligned} \text{psi } [M [\text{condarg }]] \\ =: I [(\text{app} , \text{cond} , \text{apply} , (\text{sec} , \text{distfunc}) , \text{distfunc})] \end{aligned}$$

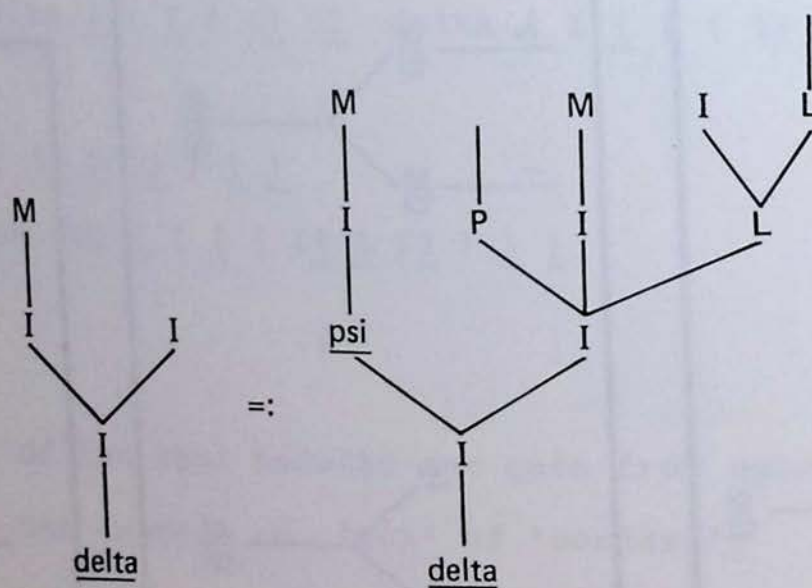
Appendix III



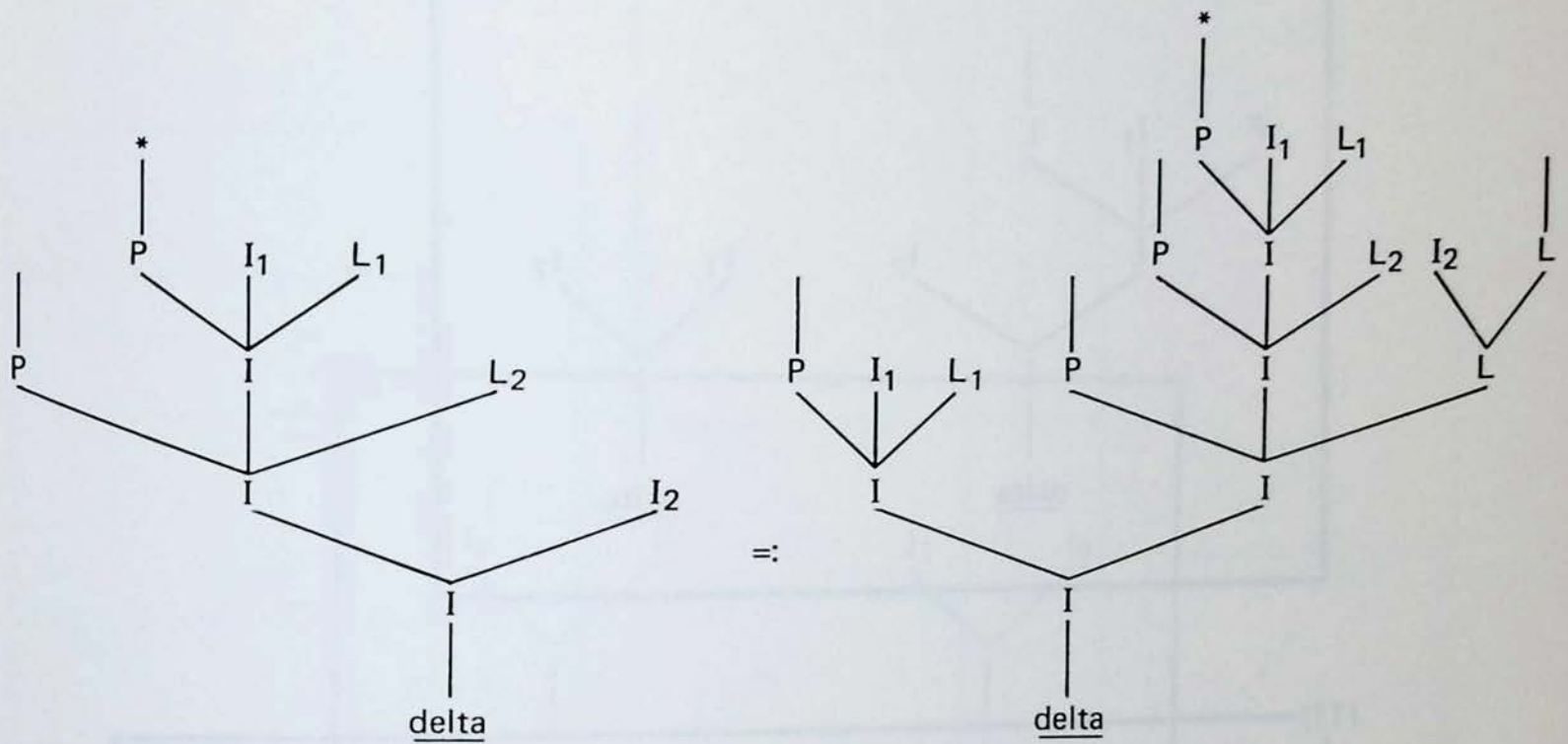
6)



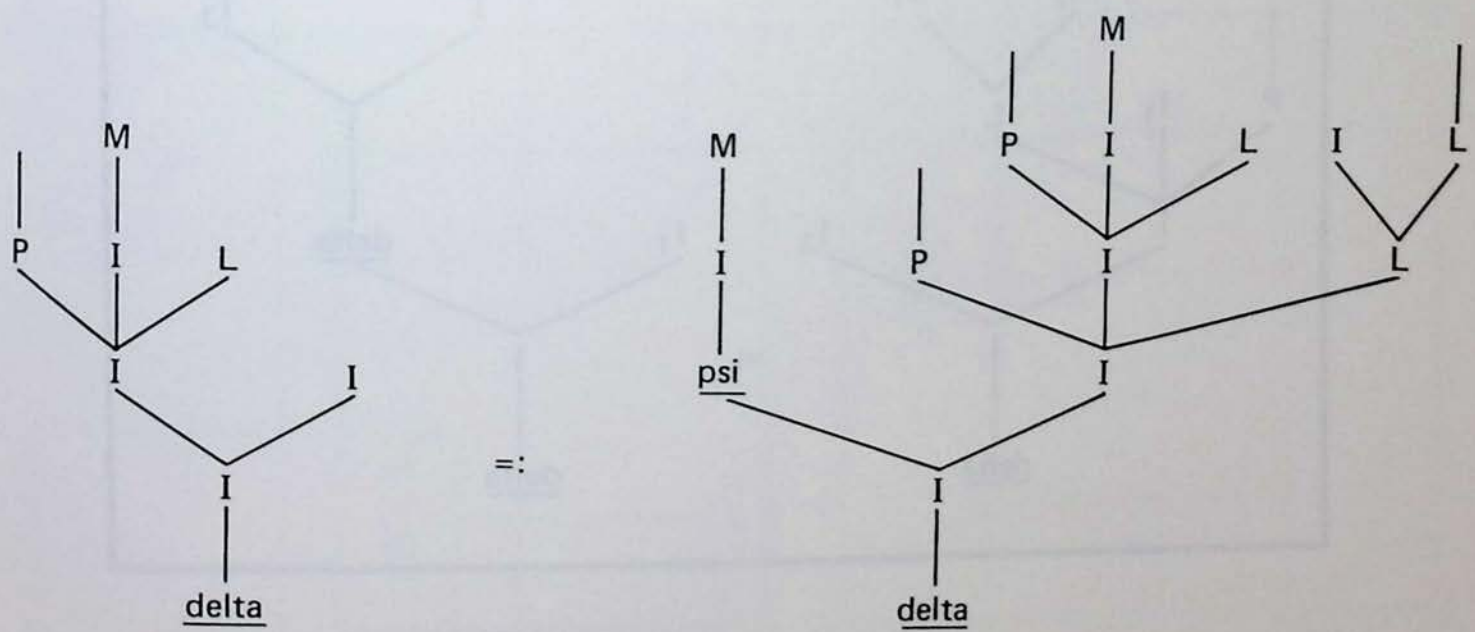
(7)



(8)

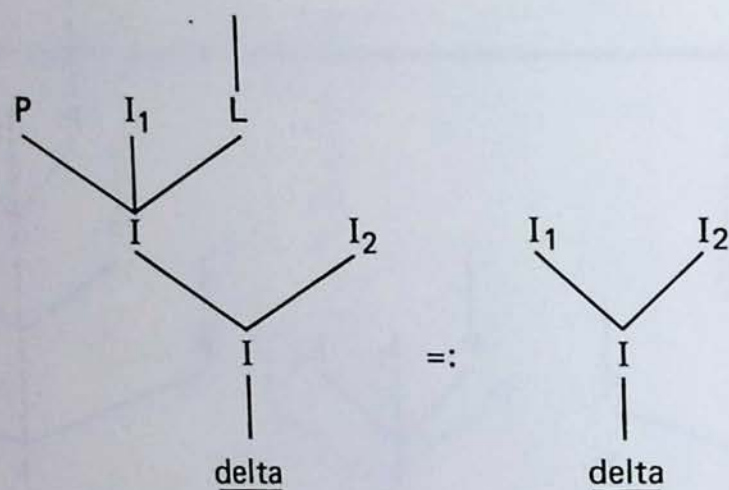


(9)

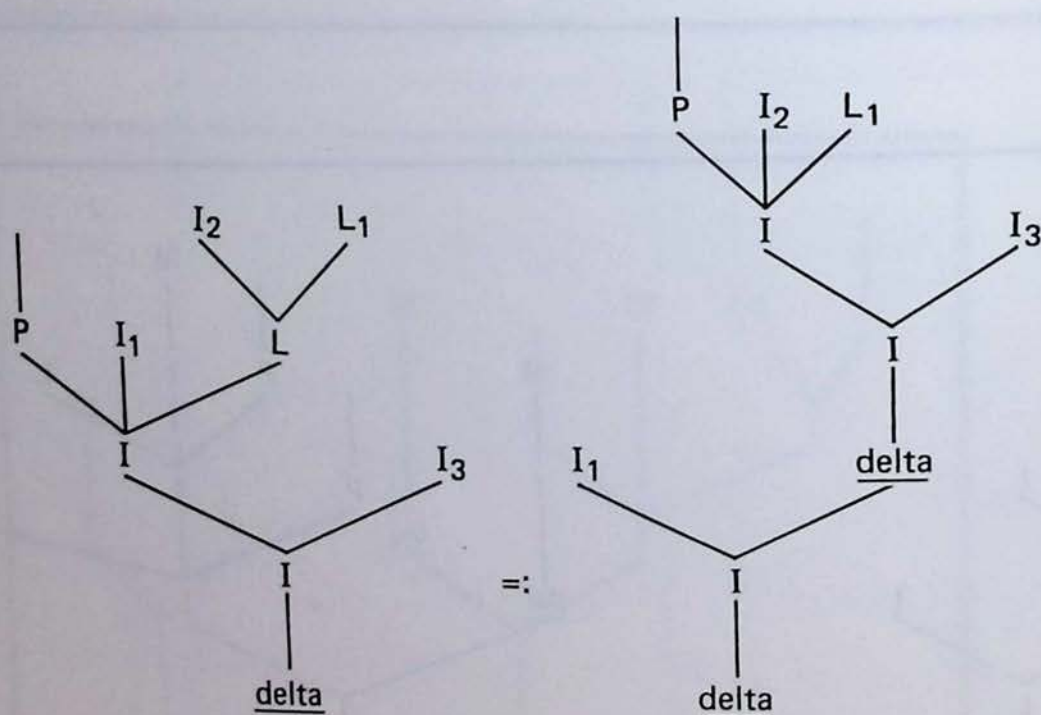


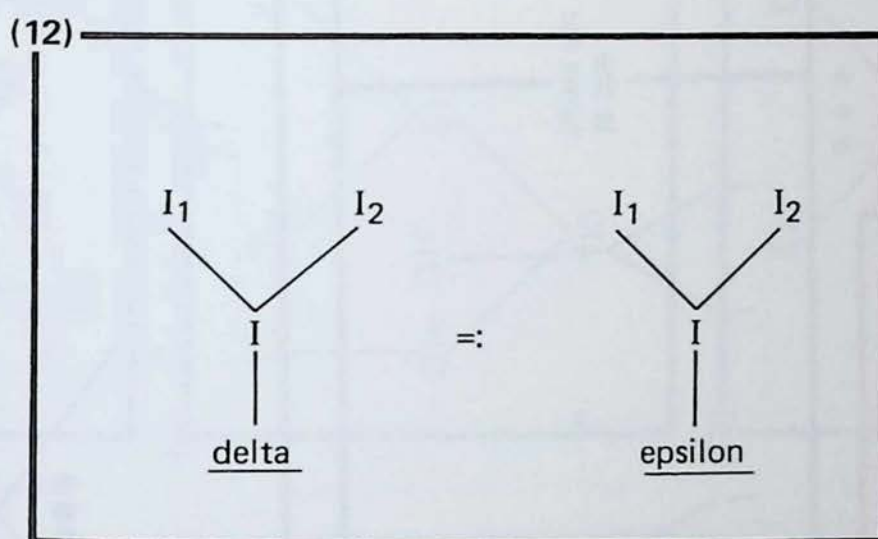
Appendix III

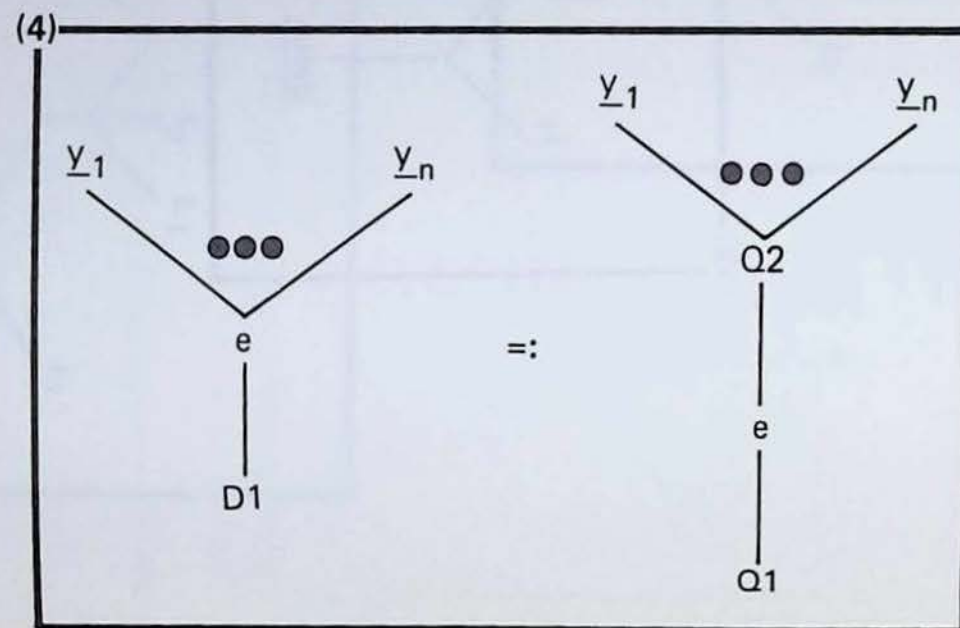
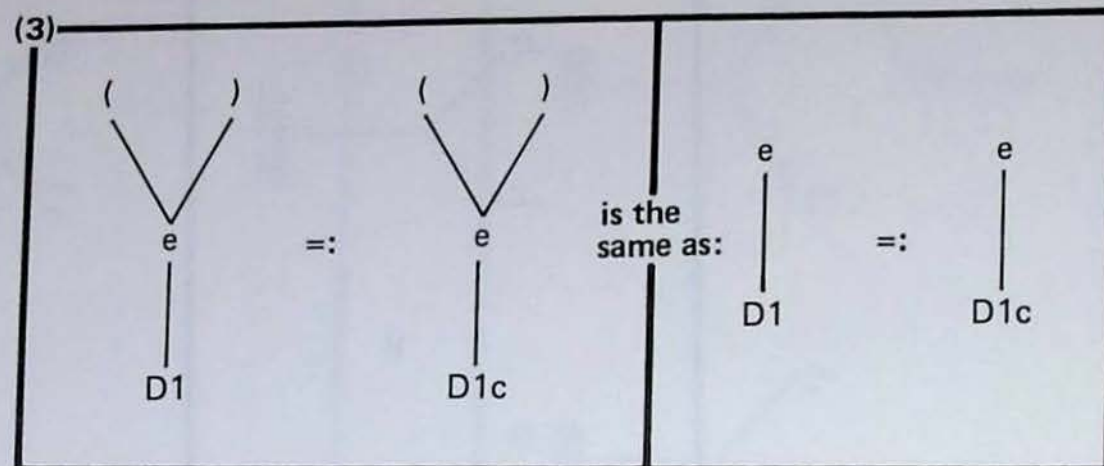
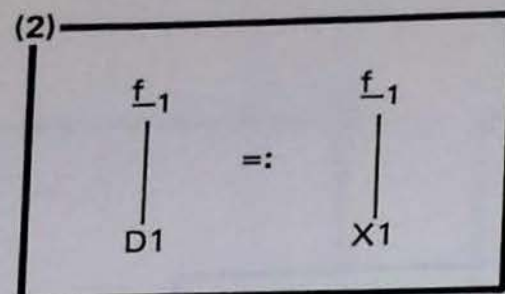
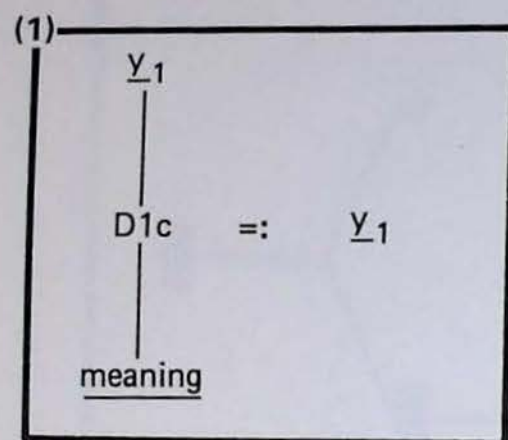
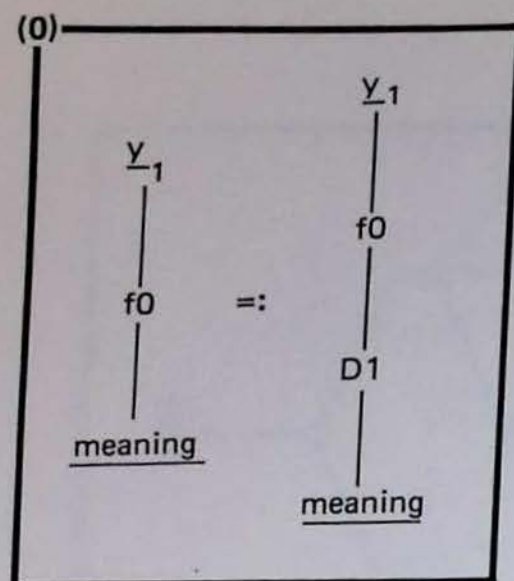
(10)

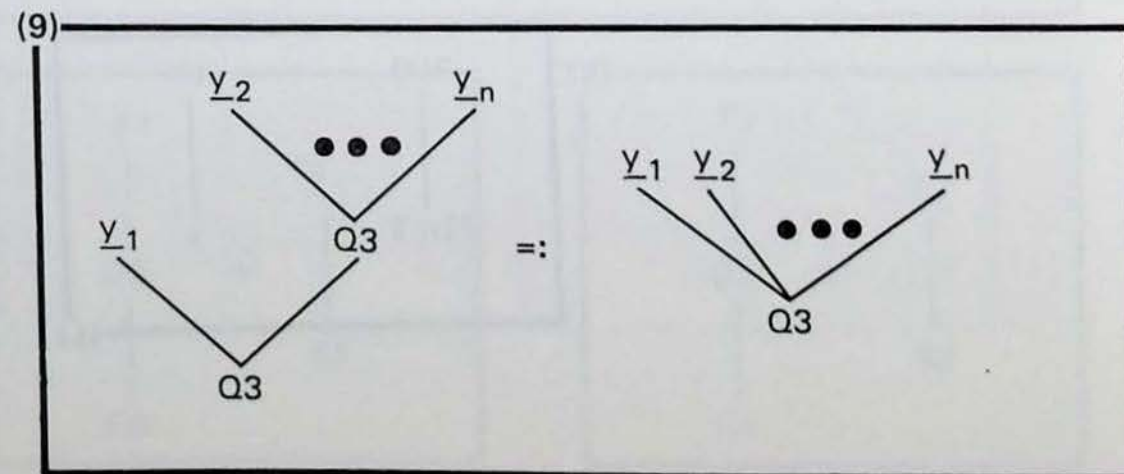
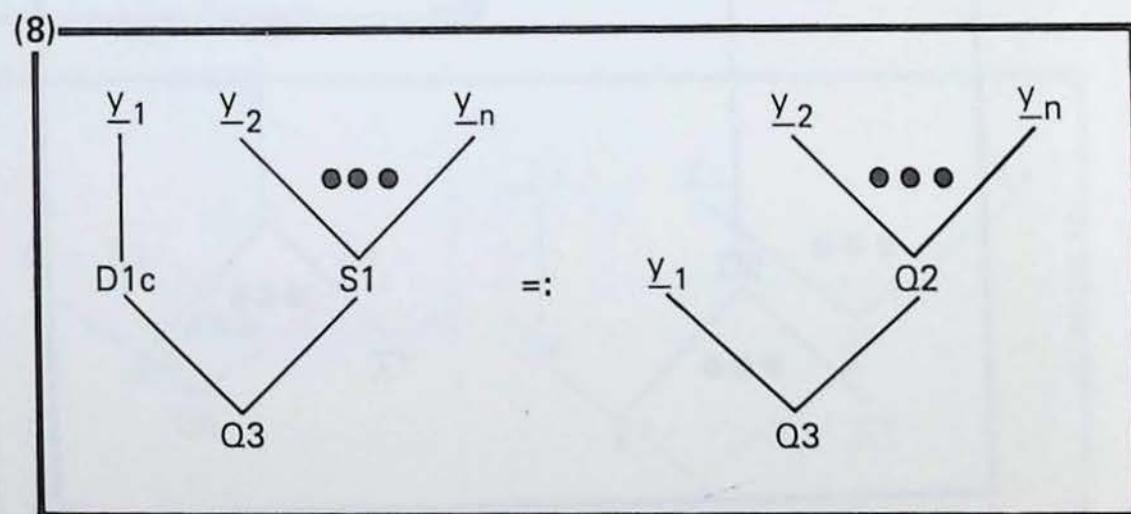
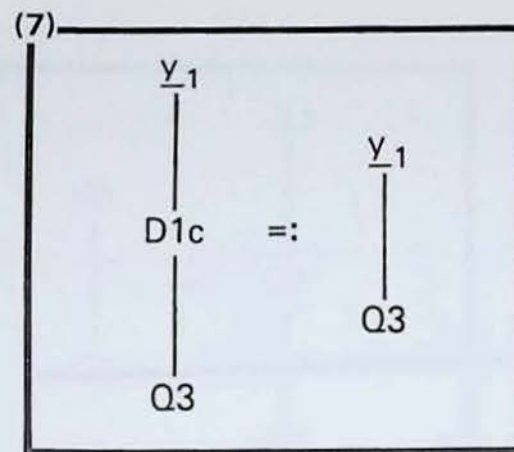
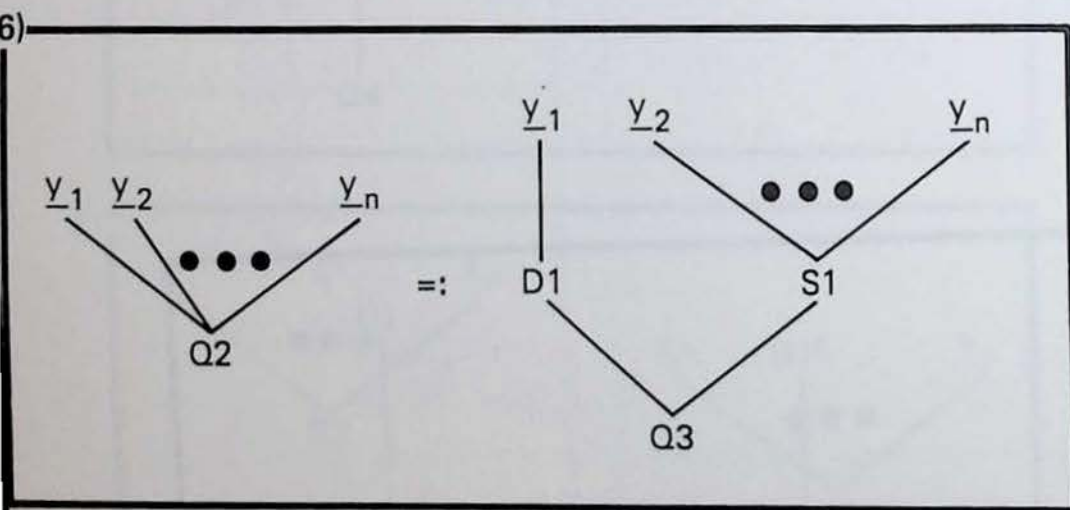
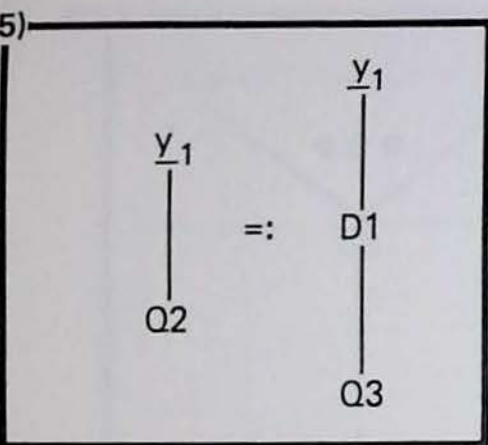


(11)

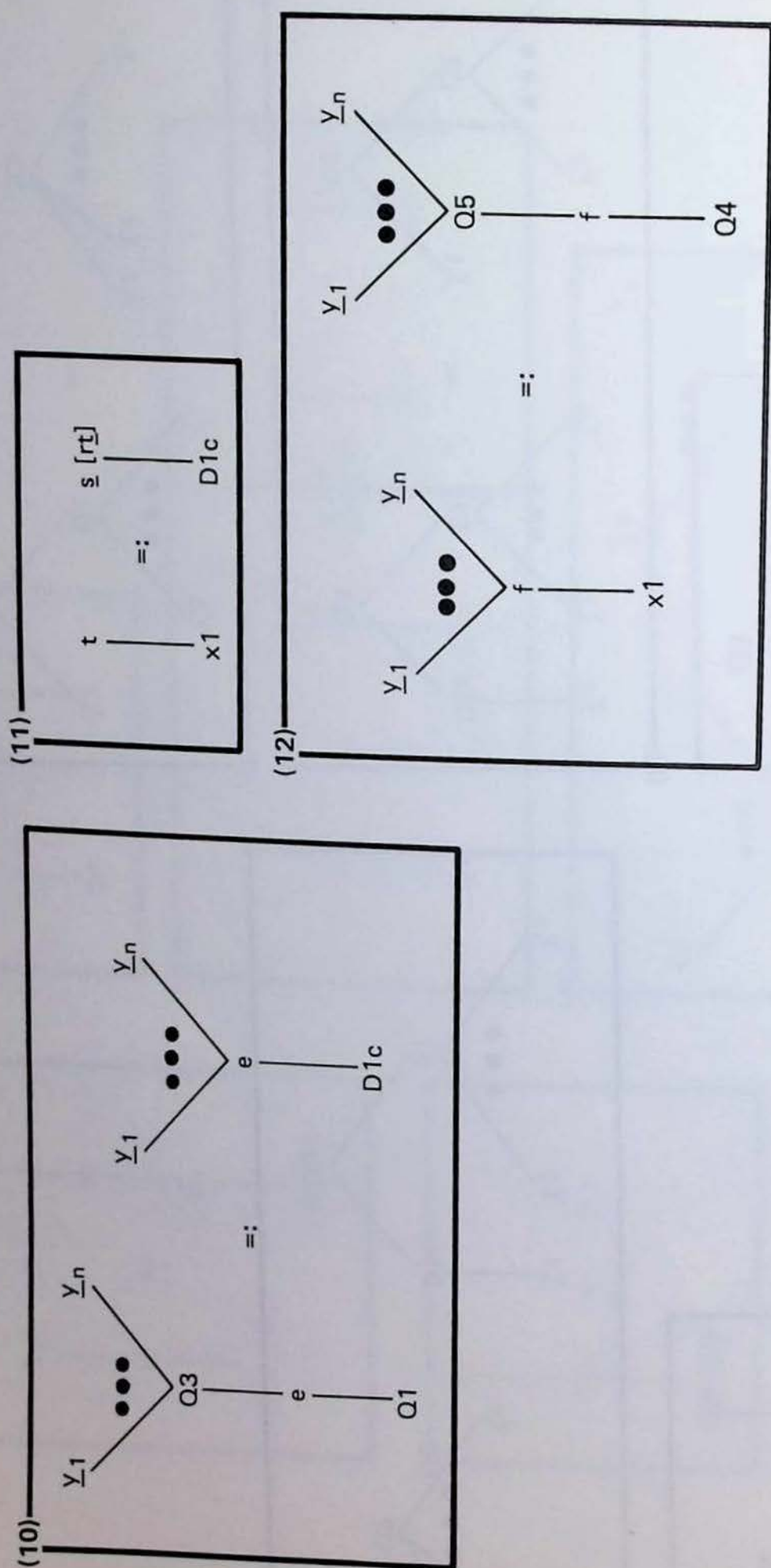


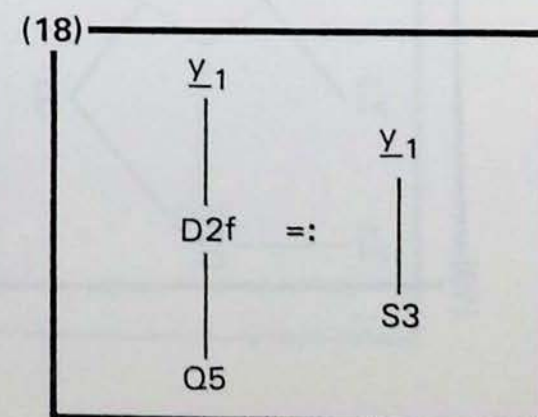
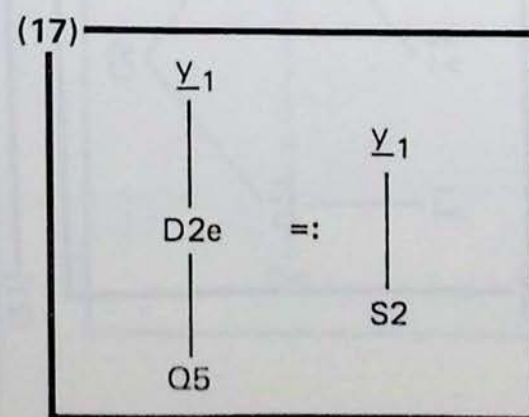
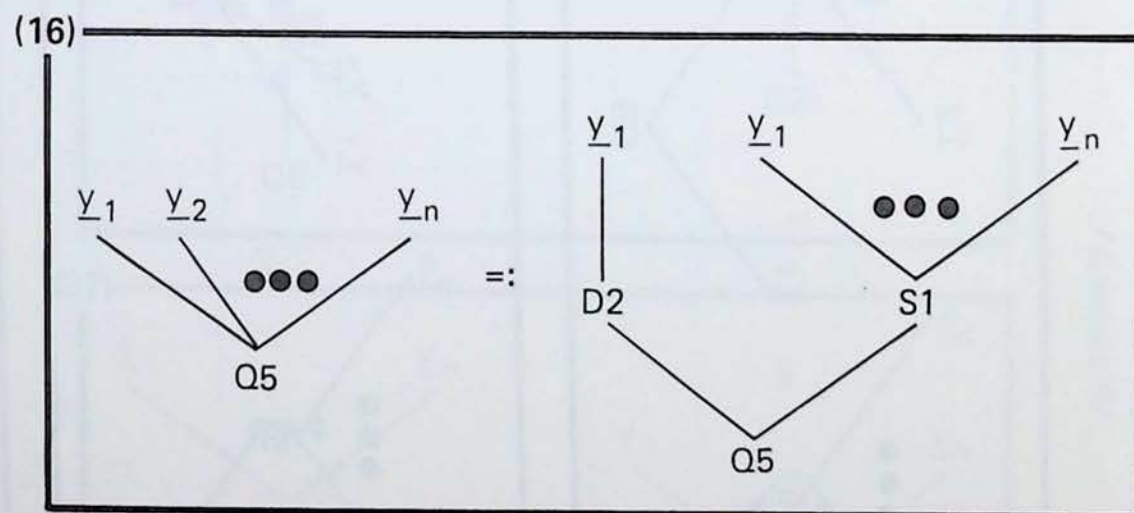
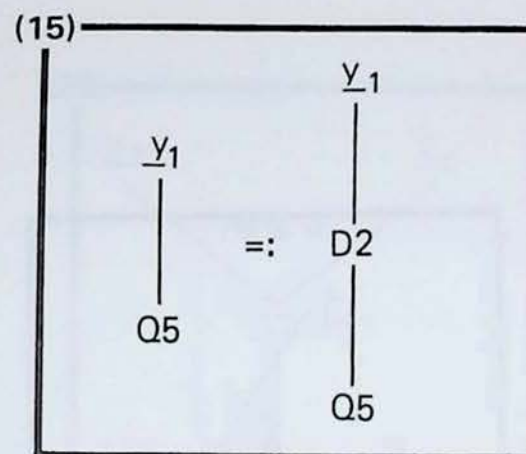
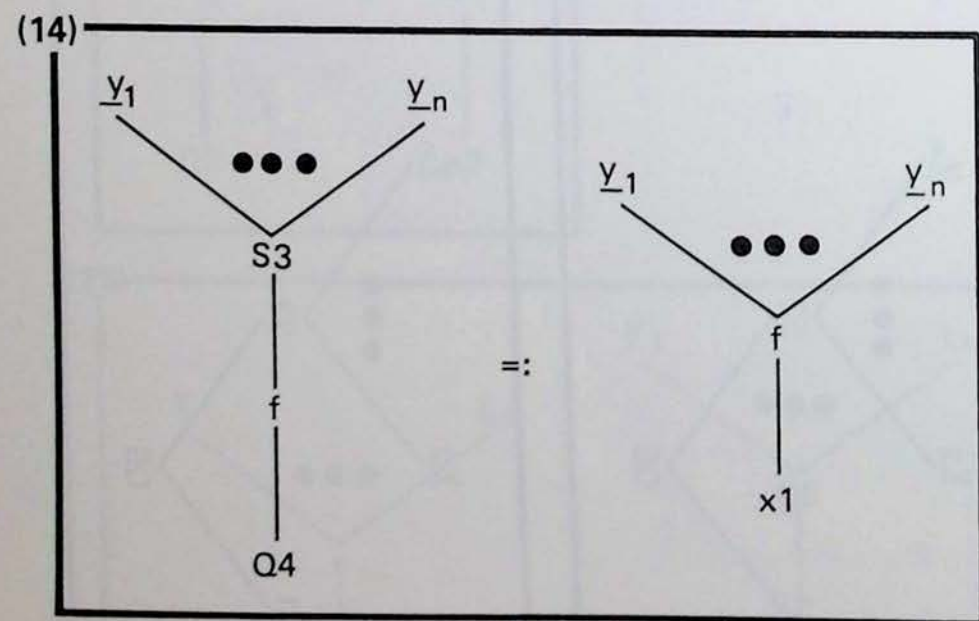
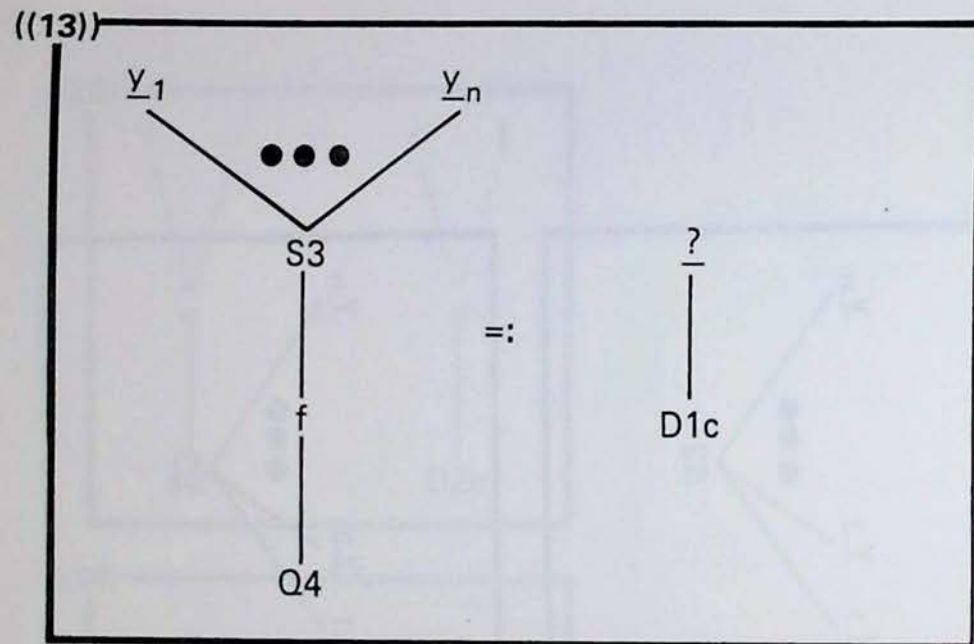






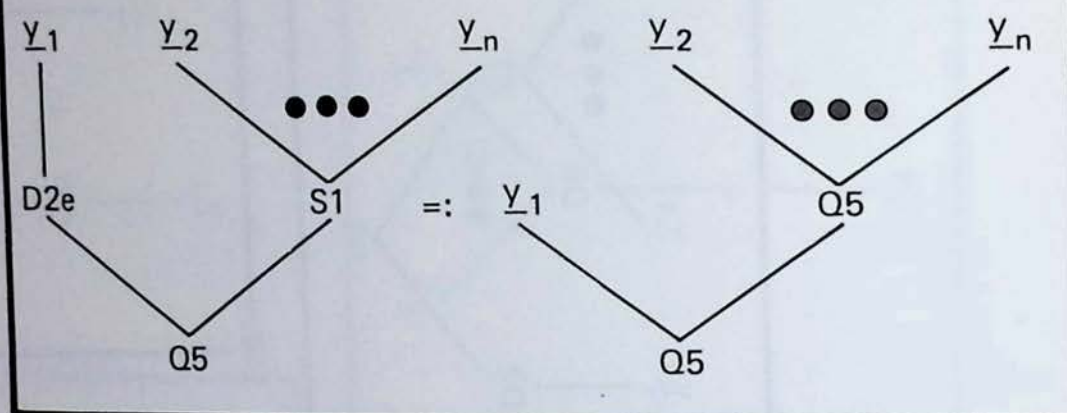
Appendix IV



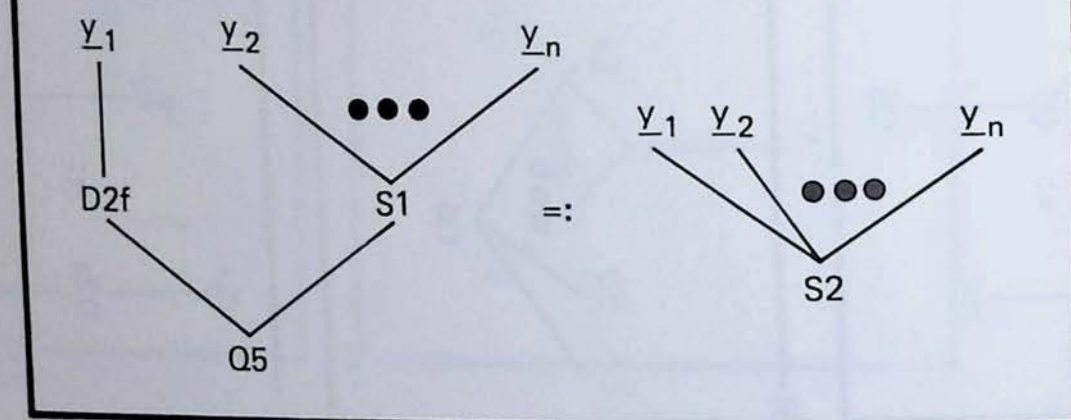


Appendix IV

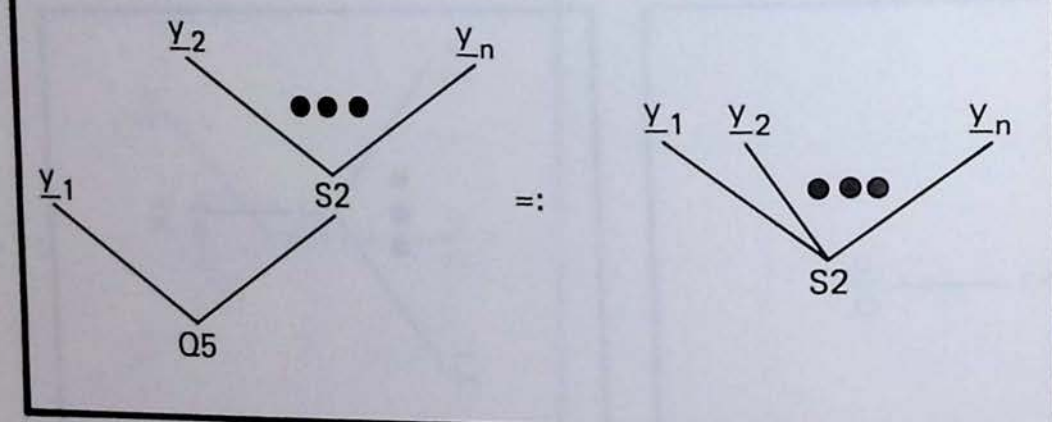
(19)



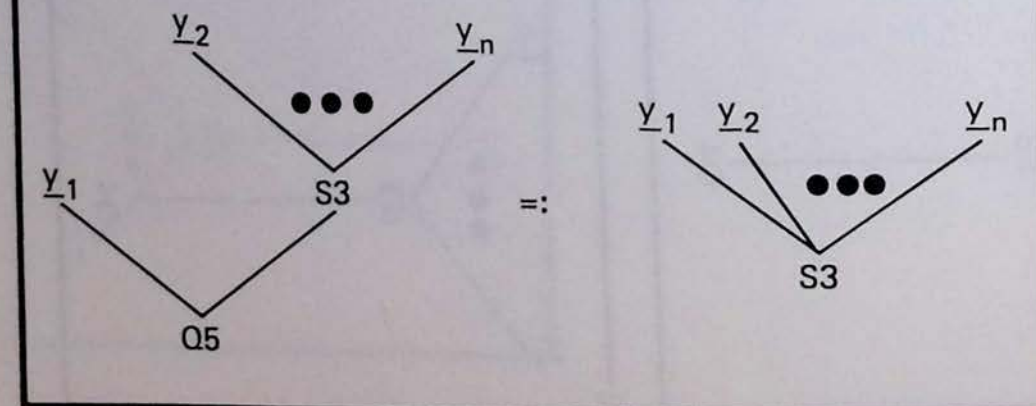
(20)



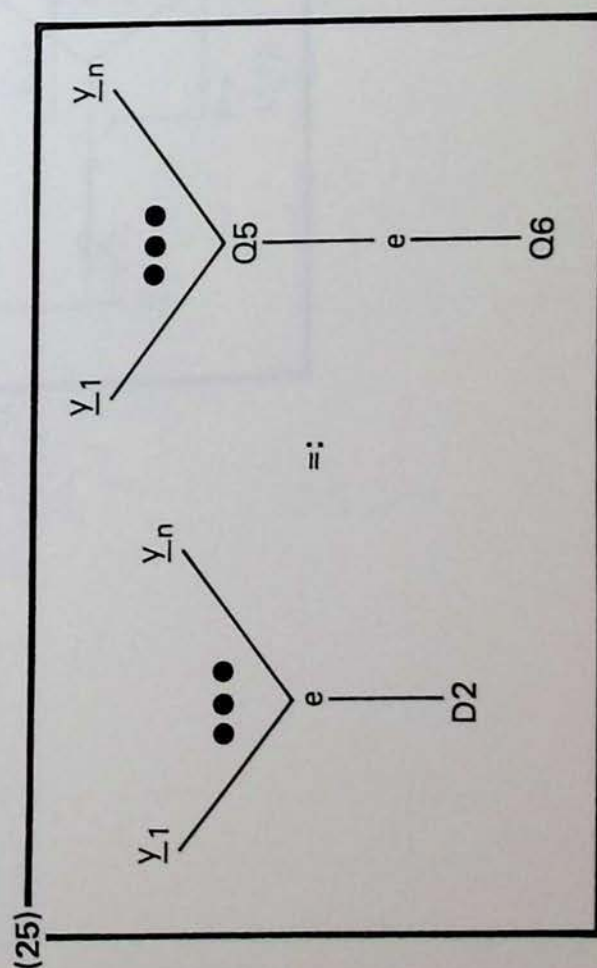
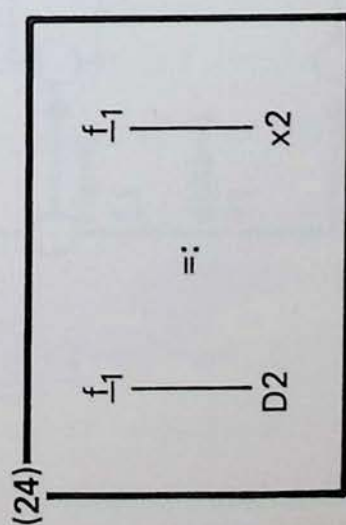
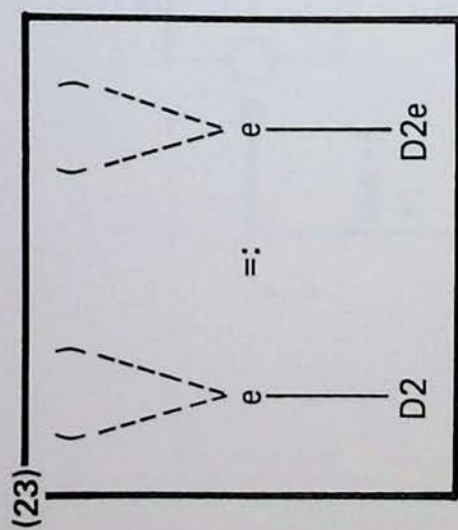
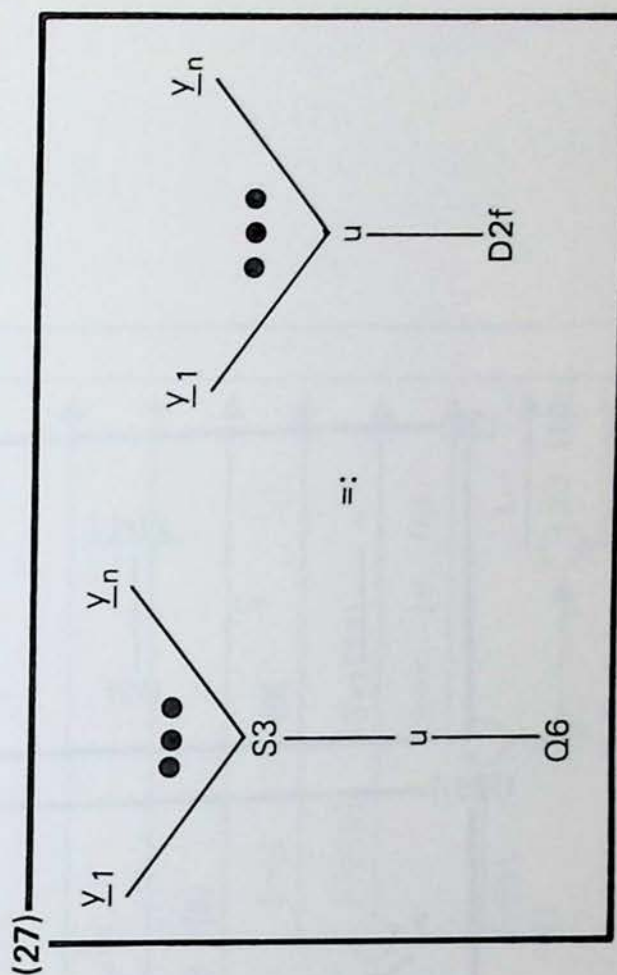
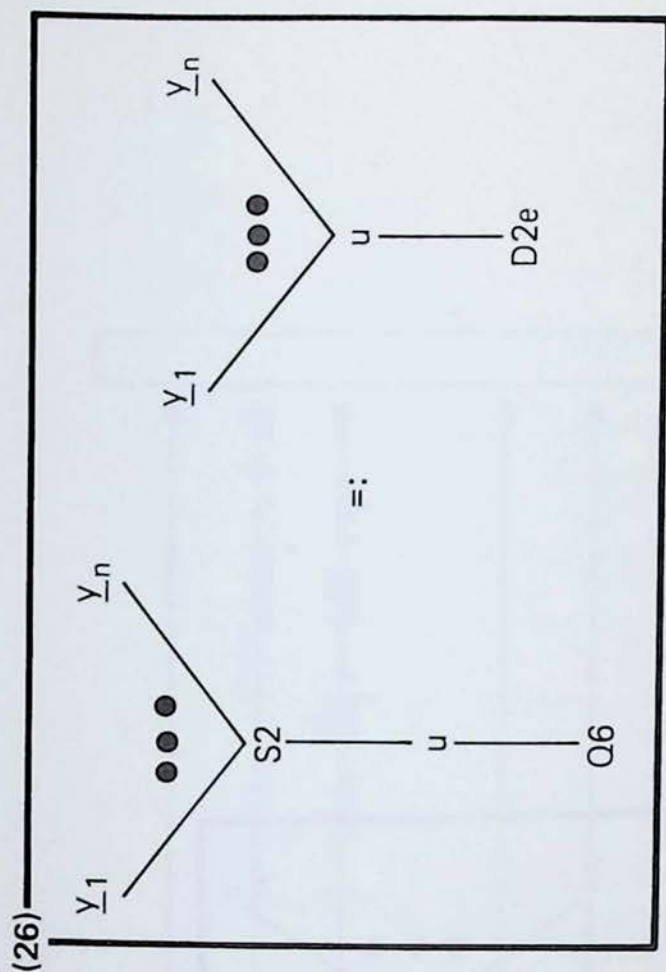
(21)



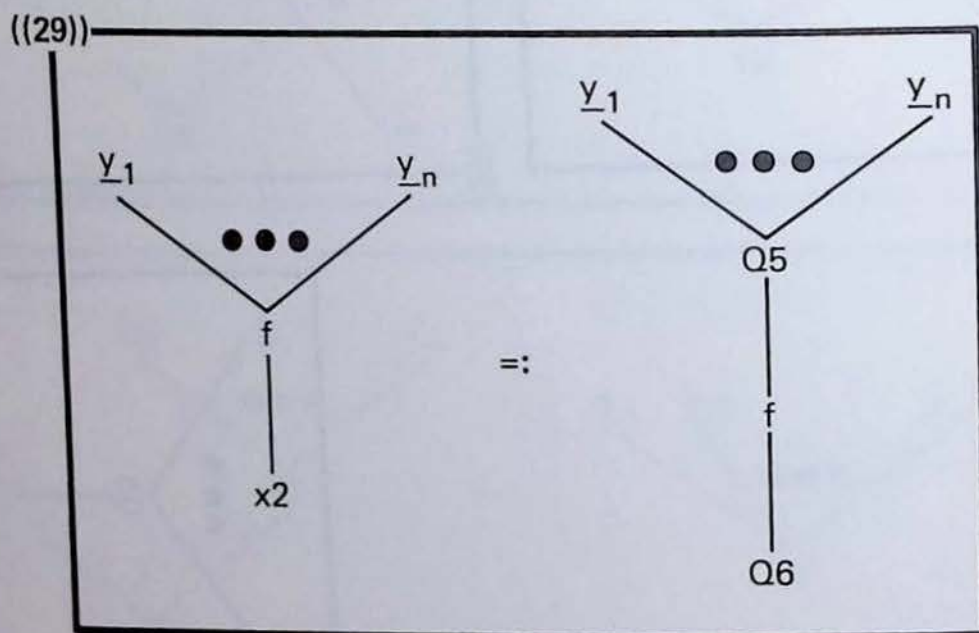
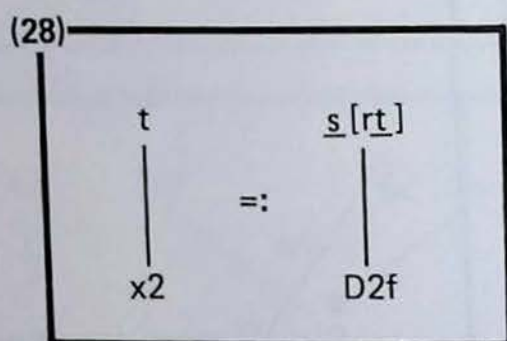
(22)

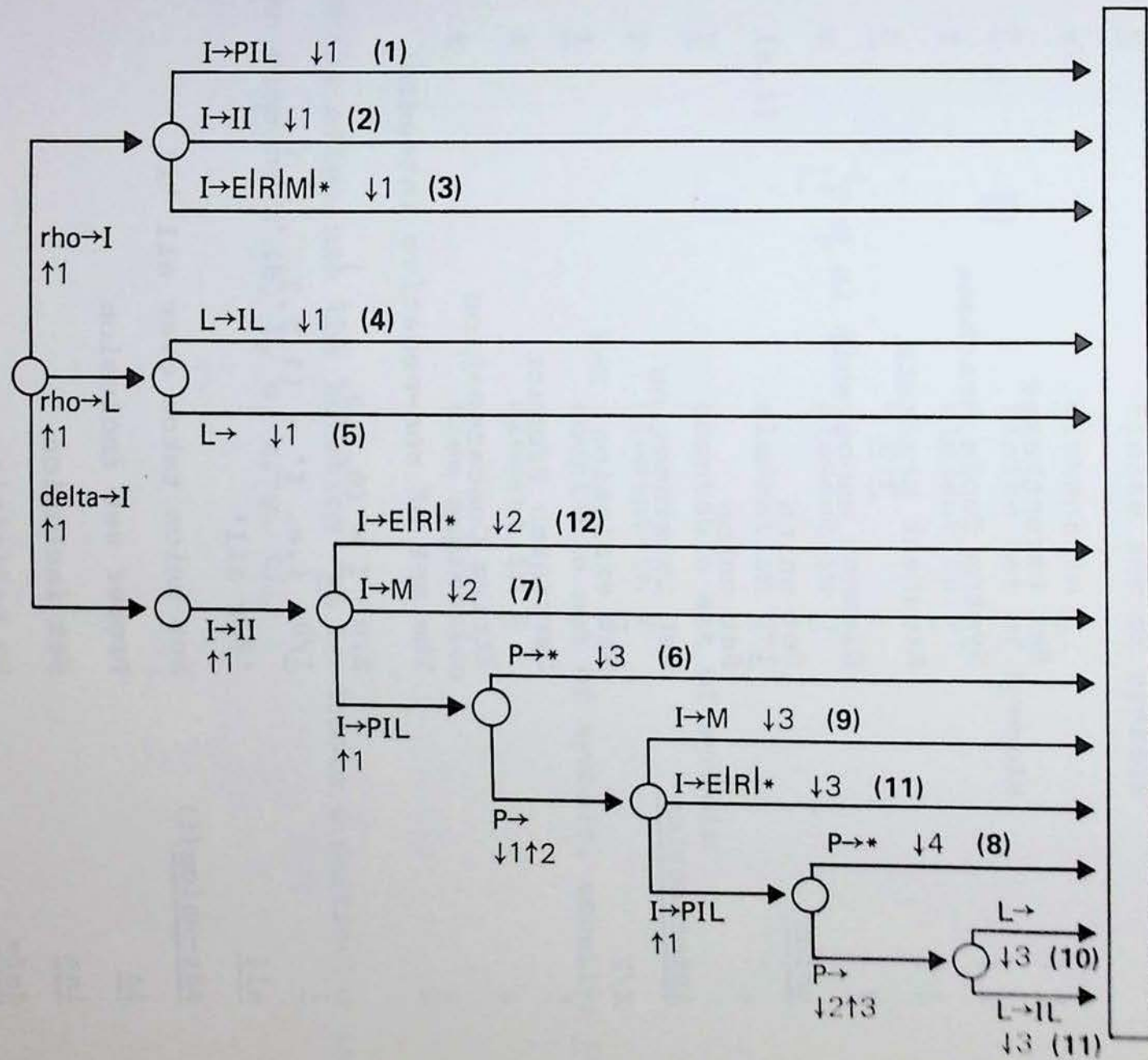


Appendix IV



Appendix IV





NOTATION

{ }	Set Parentheses
[]	System Tuple Brackets
[]	Argument Brackets
e	Element membership in Set
<u>union</u>	Set union
+	Set union
<u>intersection</u>	Set intersection
X/Y	Set exclusion, X-Y
*	Cartesian Product
*	String Concatenation
<u>I</u>	The set of non-negative integers i.e. $\underline{I} = (0, 1, 2, \dots)$
<u>I'</u>	$\underline{I}/0$, i.e. $\underline{I}' = (1, 2, 3, \dots)$
<u>all</u>	'for all'
<u>set-union(i)</u>	Set union taken over all 'i'
<u>in</u>	Proper set inclusion
<u>in=</u>	Set inclusion
<u>def=</u>	By Definition
<u>closure</u>	Transitive Closure
f.s.t.	'finite state tree-'
fstt	f.s.t. transition

FSTT	f.s.t. transformation (system)
<u>FSTT</u>	the set of all FSTTs
FSTE	f.s.t. evaluator
<u>FSTE</u>	the set of all FSTEs
<u>E</u>	finite set of symbols
e	element in <u>E</u>
<u>F</u>	finite set of symbols
f	element in <u>F</u>
<u>U</u>	= <u>E+F</u>
u	element in <u>U</u>
[u,i]	element in <u>U*I</u>
<u>Y</u>	countable set of symbols
y	element in <u>Y</u>
<u>Z</u>	countable set of symbols, usually: <u>Z=Y+U*I</u>
z	element in <u>Z</u>
t	tree expression

We shall often use the notation $\underline{x}_i$ to denote a particular instance of an element x (e, f, u, z, y, t).

1.4.1. Transmittance (T₀)

0.95

1.4.2. Absorbance (A₀)

0.02

1.4.3. Optical Density (OD)

0.02

1.4.4. Extinction Coefficient (ε)

0.02

1.4.5. Molar Absorptivity (ε_m)

0.02

1.4.6. Specific Absorptivity (ε_s)

0.02

1.4.7. Mass Absorptivity (ε_m)

0.02

1.4.8. Optical Density (OD)

0.02

1.4.9. Optical Density (OD)

0.02

1.4.10. Optical Density (OD)

0.02

1.4.11. Optical Density (OD)

0.02

1.4.12. Optical Density (OD)

0.02

1.4.13. Optical Density (OD)

0.02

1.4.14. Optical Density (OD)

0.02

1.4.15. Optical Density (OD)

0.02

1.4.16. Optical Density (OD)

0.02

1.4.17. Optical Density (OD)

0.02

1.4.18. Optical Density (OD)

0.02

1.4.19. Optical Density (OD)

0.02

1.4.20. Optical Density (OD)

0.02

1.4.21. Optical Density (OD)

0.02

1.4.22. Optical Density (OD)

0.02

1.4.23. Optical Density (OD)

0.02

1.4.24. Optical Density (OD)

0.02

1.4.25. Optical Density (OD)

0.02

1.4.26. Optical Density (OD)

0.02

1.4.27. Optical Density (OD)

0.02

1.4.28. Optical Density (OD)

0.02

PAGE: LINE: CORRECTION: or DELETION:

i3	6	CONTENT / CONTENTS .
6	11	thuogh / though
	18	meaning. / meaning [Ref. 21].
7	10	to far / too far
	10	the aspects / those aspects
13	14	machines / machine
15	5	strings or / strings of
20	6	heigth / height
	10	examples,
22	8	= t / = { t }
23	4	= u(<u>t</u> ₁ , ..., <u>t</u> _n) / { u(<u>t</u> ₁ , ..., <u>t</u> _n) }
25	8	in I / in <u>I</u>
30	11	l <u>t</u> if this / l <u>t</u> in this
36	<u>3</u>	f <u>T</u> (<u>E</u>) _m i.e. / f <u>T</u> (<u>E</u> +f)
38	6	
	14	10 / 10, part II,
40	9	subsituted / substituted
41	6	ohove / above
43	3	defs. / examples
44	<u>4</u>	def. / ex.
45	5	def. / ex.
	12	def. / ex.
46	<u>6</u>	def. / ex.
47	<u>1</u>	defs. / exs.
48	<u>7</u>	defs. / exs.

PAGE:	LINE:	CORRECTION:	or DELETION:
50	7	. <u>lt</u> 1. / .	
	<u>1</u>	<u>heigth</u> / <u>height</u>	
53	5	def. / ex.	
54	7	def. / ex.	
	9	31 / 30	
56	13	def. / ex.	
57	11	defs. / exs.	
	<u>1</u>	((6),(12) / ((6),(12))	
60	2	homomorphism / homomorphism	
61	<u>2</u>	15 / 21	
67	7		(seeming)
71	6	fstt <u>j</u> / fstt <u>k</u>	
72	2		(or fstt2 in fstt1)
76	3	u / f	
77	10	allways / always	
	<u>7</u>	premise / premiss	
	<u>3</u>	premise / premiss	
79	<u>2</u>	the tree above the / a subtree whose	
	<u>1</u>		, which
81	<u>6</u>	to / of	
	<u>1</u>	<u>up</u> -[k] / <u>up</u> [k]	
91	<u>4</u>	p1] / p1]]	
	<u>4</u>	pp] / pp]]	
	<u>2</u>	p] / p]]	
93	<u>4</u>	largerer / larger	
	<u>3</u>		are deleted,

PAGE:	LINE:	CORRECTION:	or DELETION:
93	<u>22</u>	is / are	
94	10	(<u>f</u>) / (f)	
	12	pk1 and:=pkr / pk1 ::= pkr	
	12	<u>P</u> : where / <u>P</u> and	
	<u>2</u>	<u>UP</u> [p]* <u>down</u> -1 / <u>down</u> -1* <u>UP</u> [p]	
95	11	asscoiated / associated	
97	9	<u>down</u> [k] / <u>down</u> -k	
101	<u>0</u>	/ : Levy, Leon S.: 'Tree Automata and Adjunct Grammars', The Moore School of Electrical Engineering, University of Pennsylvania, Interim Technical Report, 72-16, 1972.	
	15	: / : 'Definitional Interpreters for Higher-Order Programming Languages',	

LEGEND:

(underlined integers) denote line numbered from bottom of page,
otherwise line numbered from top of same.

'txt1' / 'txt2' : replace 'txt1' with 'txt2', 'txt1' may be
null.

