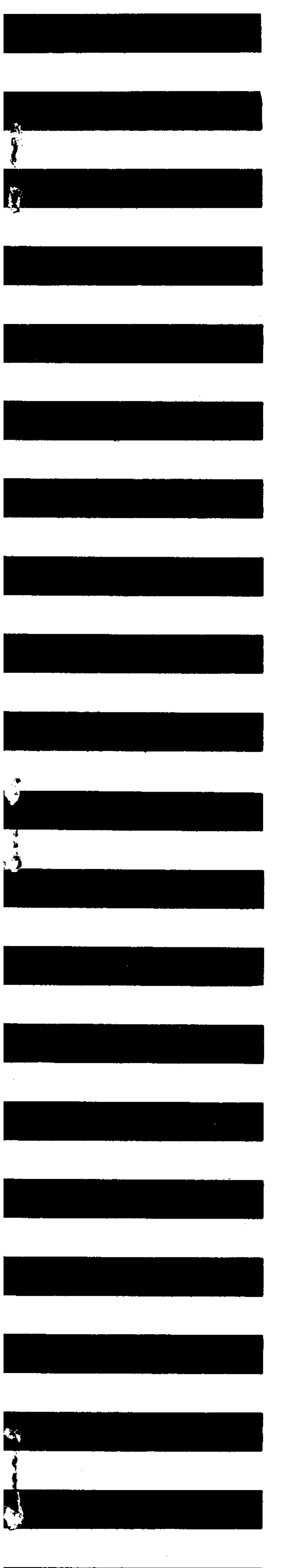


Department of Applied Analysis
and Computer Science

University of Waterloo

Technical Report CS-73-08

April, 1973

A FORMAL DESCRIPTION OF ALTRAN USING LINKED FOREST MANIPULATION SYSTEMS

by

Mansour Farah

Faculty of Mathematics

University of Waterloo

Waterloo, Ontario

Canada

Department of Applied Analysis

&

Computer Science

Department of Applied Analysis
and Computer Science

University of Waterloo

Technical Report CS-73-08

April, 1973

A FORMAL DESCRIPTION OF ALTRAN USING LINKED FOREST MANIPULATION SYSTEMS

by

Mansour Farah

This research was supported by the National Research Council of Canada, Grant No. A-7403.

ABSTRACT

The syntax and semantics of a programming language, ALTRAN (with minor modifications) which is mainly designed for algebraic manipulations, are completely formalized.

A model for formal definition of programming languages, based on the notion of Linked Forest Manipulation System, is used for the purpose of this formal description.

ACKNOWLEDGEMENT

The author wishes to express his gratitude to his supervisor, Dr. K. Culik for suggesting the topic and for his guidance throughout the course of this project.

TABLE OF CONTENTS

	Page
INTRODUCTION	1
CHAPTER 1 - THE MODEL FOR FORMAL DEFINITION	3
1. Linked Forest	4
2. Tree Production	4
3. Programmed Grammar	5
4. Linked Forest Manipulation System	6
5. Macro-Operations on Trees	11
6. Model for Language Description	11
CHAPTER 2 - OVERVIEW OF THE ALTRAN PROGRAMMING LANGUAGE	13
1. Constants and Variables	14
2. Substitution	15
3. Invocation	15
4. Expressions	16
5. Statements	16
6. Procedure	17
7. Program	17
8. The Data Language	17
9. Restrictions on ALTRAN	18
10. Example	18
CHAPTER 3 - FORMAL DESCRIPTION OF ALTRAN	20
I. The Syntax Description	24
1. Basic Tokens	24
1.1 Letters and Digits	24
1.2 Identifiers	24
1.3 Arithmetic Constants	25
1.3.1 Integer constants	25
1.3.2 Signed integer constants	26
1.3.3 Real constants	27
1.4 Logical Constants	30
2. Attribute statements	31
2.1 Attributes	31
2.2 The attribute statement	34
2.3 List of attribute statements	37
3. Arithmetic, logical and label expressions	38
3.1 Arithmetic expressions	38
3.2 Logical expressions	44
3.3 Label expressions	46

TABLE OF CONTENTS (cont'd)

Page

4. Statements 47
   4.1 Elementary statements 47
      4.1.1 The assignment statement 47
      4.1.2 The return-statement 47
      4.1.3 The goto-statement 47
      4.1.4 The continue, doend and end statements 47
      4.1.5 The Input/Output statements 47
   4.2 Invocation statements 50
      4.2.1 Dummy and Actual Variables 50
      4.2.2 Invocation statement 50
      4.2.3 Algebraic options statement 50

5. Groups
   5.1 The do-group 53
   5.2 The if-group 53
   5.3 Labeled and unlabeled group - Body 53

6. Procedure 58
   6.1 Main procedure heading 58
   6.2 Procedure heading 65
   6.3 Procedure 66
      6.3.1 Procedure structure 66
      6.3.2 Compatibility of types in an arithmetic expression 72
      6.3.3 Compatibility of types in the assignment statement 76
      6.3.4 Some restriction related to dummy variables 77

7. Program 80

8. The Data Language 84

9. Job 88

II. The Semantics Description 89

1. Initialization 89
2. List of statements 95
3. The continue, goto, if, doend and end statements 95
4. The return statement 97
5. Element Accessing 101
6. Logical operations 103
   6.1 Logical assignment 103
   6.2 Logical expressions 104
7. Label assignment 106

TABLE OF CONTENTS (cont'd)

	Page
8. The Arithmetic Assignment Statement	110
8.1 Conversion of precision	110
8.2 The Conversion of type	113
8.3 The assignment of a value	116
9. Arithmetic expressions	123
9.1 Evaluation of constants and variables	123
9.2 Substitution evaluation	127
9.3 Evaluation of addition, subtraction, multiplication and division	134
9.4 Evaluation of exponentiation	139
9.5 Conversion of types in expressions	143
9.6 Overflow	145
10. Relation - Comparison	146
11. Algebraic options statement	148
12. Procedure invocation	148
12.1 The invocation statement	148
12.2 Macro inbound	149
12.3 Macro outbound	151
12.4 Arrays in inbound and outbound transmission	153
13. The Input/Output statements	154
REFERENCES	157
INDEX	158

INTRODUCTION

The complete formalization of a programming language has been felt as a real need by both the designers and the users of programming languages. It was felt that several problems that those working on or using programming languages encounter, would be resolved if a reasonable formalization of the syntax and semantics can be made (e.g., proofs on programs and in particular proofs of correctness of compilers).

Several kinds of formalisms have been found. None of them however is perfect, none of them satisfies both the user and the designer of programming languages.

One of the latest formalisms that deal with this subject is due to K. Culik [2] and is a model based on the notion of linked forest manipulation system. This model has the advantage of having simple and elementary basic tools that are trees, pointers and labels. It can be considered as an operational model, i.e. a model that is not purely descriptive but that gives a way to construct a compiler. Although this model seems interpretative, it can be used as a basis for either an interpreter or a compiler.

In this work a programming language for algebraic manipulations, ALTRAN (for ALgebra TRANslator), is described using this model for formal definition. The model, as we said, is quite simple as far as the basic tools are concerned. However, the ALTRAN language as it is designed is not very simple. That is why some features of the language have been changed or deleted to keep the description in some reasonable bounds.

Before giving the formal description of ALTRAN, a review of the method for formal definition of programming languages described in [2], is given. This constitutes Chapter 1. Chapter 2 is an overview of the ALTRAN programming language as it is described in [1]. Some restrictions and changes made on the language are listed in this same chapter. Chapter 3 gives the formal description of ALTRAN with the modifications given in Chapter 2.

CHAPTER 1

THE MODEL FOR FORMAL DEFINITION

This model [2] can be viewed as a machine formed of two parts. The first part is a generator , the second an interpreter. The _generator_ part describes the syntax of the language by generating two distinct objects, a concrete program and an abstract program.

The _concrete program_ is a string which constitutes a program in the given language. A context-free grammar generates this concrete program.

The _abstract program_ is a labeled, ordered, rooted forest with pointers, that represents the program on an abstract level. This forest is built up tree by tree in parallel with the generation of the concrete program. Some transformations are eventually made on a tree, or forest, using a linked forest manipulation system (l.f.m.s.) that is described below.

This generator, as it is described, defines a translation from concrete programs to abstract programs. Thus it can be used as a model for building a translator.

The _interpreter_ part executes the abstract program statement by statement. This interpreter is nothing but a linked forest manipulation system that performs some transformations on the abstract program (a linked forest) and yields the resulting linked forest on which the results of execution can be found.

1. Linked Forest

Different formalisms for defining a tree exist. One of them is given in [2], but they are not of great interest for this work.

The notion of tree being understood, a labeled tree is a tree whose nodes may be labeled or multi-labeled (i.e. several labels are attached to a given node). These labels can be chosen from a given set of labels.

A finite collection of labeled trees (forest), that are ordered and such that pointers (or links) may exist between their nodes, is called a linked forest.

A pointer can be considered as a member in some relation on the nodes of a forest which is different from the relation defining the edges of a tree. Thus a pointer is a pair of nodes that will be represented by an arrow between two nodes on any graphic representation of a tree.

2. Tree Production

A tree production defines a transformation that can be performed on trees or, more generally on forests. Such a production has a left-hand side and a right-hand side, both being trees or forests. When a tree production is applied to a tree or forest that contains a part "similar", in respect to some well defined rules, to the left-hand side of the production, it yields a transformed tree having a part "similar" to the right-hand side of the production.

The transformation involved in such a production consists, in general, of some subtree replacement. This kind of replacement is not difficult to define precisely when the trees have no pointers. But in the case of linked forests we need to be much more careful for we have to specify exactly all what happen to the links with the environment whenever a subtree is replaced by another one.

That is why a special kind of nodes that can appear in a tree production is distinguished. These nodes are called pivotal nodes. A pivotal node (double circled in any graphical representation) are the privileged nodes whose descendants may be replaced in a tree production. But at most one pivotal node may appear in the forest involved in a tree production and whenever it appears, must do so on the left as well as the right-hand sides.

3. Programmed Grammar

This notion is defined in [4] and is a generalization of the notion of grammar in which the usage of a production is free of any restriction. In a programmed grammar the productions (in our case the tree productions) are labeled by some production labels. Also each production has two supplementary fields that are called success and failure fields. There is a distinguished production label (START), and it is at a production labeled START that the system is entered for generation or transformation. If a production can be applied successfully, the next production to use is one labeled by the label of the success field.

Otherwise, it is one labeled by the label of the failure field that should be used.

4. Linked Forest Manipulation System (l.f.m.s.)

The notion of l.f.m.s. is essentially based on the two notions of tree production and of programmed grammar that were mentioned above.

An l.f.m.s. is a set of tree productions each of them having three supplementary fields:

i) Production label field

ii) Success field

iii) Failure field.

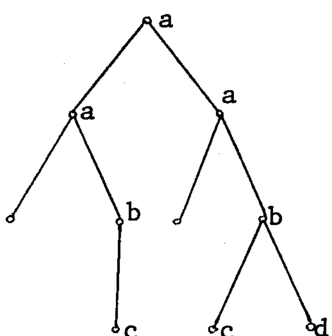
Two distinct production labels are START and STOP. It is at the production labeled START that the system is entered. The next production to use is determined, like in a programmed grammar, by the success or failure to apply the previous one. When the STOP label is encountered we go out of the system. If the STOP label is not encountered in a finite number of steps there is an error.

Example of an l.f.m.s.

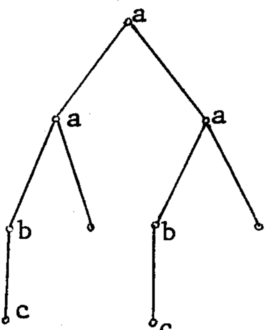
If a, b, c and d are labels the following defines an l.f.m.s.

START	a b c d	→	a b c d	START	L_1
L_1	a b c d	→	a b c	L_1	L_2
L_2	a b c	→	a b c	L_2	STOP

Note that a pivotal node is used whenever there is a transformation on the structure of the subtree attached to this node. For example, if we have the following tree:



it could be transformed using the previous l.f.m.s. into:



In fact, an l.f.m.s. is more powerful than what we have just described. Three tokens are introduced to add some power to it. They are, the label parameters, the tree parameters and the function denotations.

A <u>label parameter</u> can be considered as a variable that can be replaced by any label from a given subset of the set of labels called its domain.

A tree parameter can also be considered as a variable that can be replaced by any tree from the parameter domain which is a set of labeled trees.

Basic functions, whose formal definition is supposed to be known, have a domain which is a subset of the set of inputs (that includes all labels, basic numbers and symbols), and may be composed to form what is called a function denotation. These function denotations may be used as any label or label parameter but only on the right-hand side of a production.

When using in a tree production such tokens as label parameters, tree parameters or function denotations, we have a production that is equivalent to several tree productions and in general to an infinity of them. Such a generalized tree production is called a production schema on trees. Thus an l.f.m.s. in all its generality is a collection of production schemata.

Now that the form of a production schema and that of an l.f.m.s. are described, we are going to describe, also in an informal way, how to use such a system or more precisely to explain what does a production schema mean.

Given a linked forest and a production schema s, $s \equiv L_1 : u_1 \rightarrow u_2 \; Suc(L_2) \; Fail(L_3)$, we can map the forest w_1 into a forest w_2 using the production schema s if the forest w_1 can be separated into two forests v_1 and w_1' and the forest w_2 can be separated into two forests v_2 and w_2' such that the following conditions are respected.

(a) v_1 (resp. v_2) is obtained from u_1 (resp. u_2) by:

- replacing each label parameter in u_1 and u_2 by the same element of its domain all over u_1 and u_2.
- replacing each tree parameter in u_1 and u_2 by the same linked tree all over u_1 and u_2. This linked tree is obtained from a tree belonging to the domain of the parameter by eventually constructing pointers between the tree and w_1' for u_1 or w_2' for u_2. The order of the tree parameters and sons of any node being respected.
- replacing each function denotation in u_1 and u_2 by the element it denotes (in the set of inputs).

(b) When a node is common to v_1 and w_1', a label or pointer attached to it can belong to either v_1 (resp. v_2) or w_1' (resp. w_2') but not to both of them.

(c) If u_1 (resp. u_2) has a pivotal node, no descendant in w_1 (resp. w_2) of this pivotal node may belong to w_1' (resp. w_2').

(d) For any node labeled by tree-parameters in u_1 (resp. u_2) the only direct descendants (or sons) of this node that may belong to w_1' (resp. w_2') are those belonging to v_1 (resp. v_2).

(e) w_1' and w_2' are identical in respect to the tree structure, labeling and pointers between their own nodes.

Example: Suppose that the set of inputs is $\Sigma = \{a, b, c, d, +, -, *, /\}$ and let ν be a label parameter whose domain is $\{+, -\}$, we write $A = \{(\nu, \{+, -\})\}$, A being the set of label parameters and their respective

domains. Let $B = \{(u, \Sigma_*), (v, \Sigma_*), (w, \Sigma_*)\}$ be the set of tree parameters and their respective domains. Σ_* denotes the set of trees whose nodes are labeled with some elements of Σ. Let $F = \phi$ be the set of basic functions $J = \{START, STOP, L\}$ be the set of production labels and R be the set of production schemata that follows. $S = (\Sigma, A, B, F, J, R)$ defines an l.f.m.s.

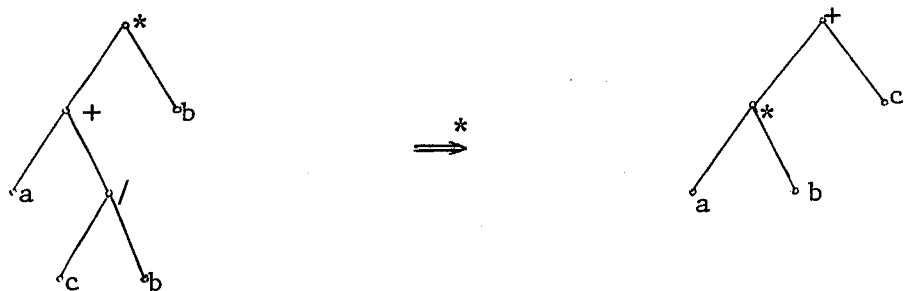
		Success	Failure
START	* ν w u v → ν * * u w v w	START	L
L	* / v u v → u	L	STOP

This system reflects the following algebraic rules:

$$(k + \ell)*m = k*m + \ell*m$$

and $$(k/\ell)*\ell = k.$$

Using this l.f.m.s. we can write the following derivation:



5. Macro-Operations on Trees

To make the l.f.m.s. more powerful some functions on trees may be defined and they are called macro-operations. A macro-operation, like any function denotation, may appear on the right-hand side of any production schema.

These macro-operations are defined using an l.f.m.s. and they yield a linked tree when given a forest as parameters. Several macro-operations that are formally described will be used in the description of ALTRAN. Sometimes when the meaning of a macro-operation can be clearly defined informally, we are omitting a possible cumbersome formal definition by an l.f.m.s.

6. Model for Language Description

Based on the notion of l.f.m.s. this model, as was mentioned in the beginning of this chapter, has two parts.

The syntax part is a set of syntax rules. Each rule is formed of a context-free production, a linked tree and eventually an l.f.m.s.

The context-free productions generate a program in the given language (concrete program) while, in parallel, the corresponding linked tree is constructed in a bottom up manner. When there is an l.f.m.s. some modifications on the structure of the constructed tree are performed. These transformations reflect in most of the cases some non-context-free properties of the language. In some other instances, it is used to make some technical changes that facilitate the description later on.

Thus the syntax part, when it is performed, constructs an abstract linked forest representing the program that it generates. It should be noted that the construction of this linked forest gives a bottom up algorithm for translation from concrete to abstract programs.

The semantics part is one l.f.m.s. which performs some prescribed transformations on the linked forest obtained from the syntax part. These transformations correspond in fact to an execution (more precisely to an interpretation) of the program. As in any l.f.m.s. besides the label and tree parameters, function denotations and macro-operations may be used to make the presentation more compact, more natural and easier to follow. For example, some technical details may be left apart by defining a macro-operation.

When the l.f.m.s. stops (and if it does so) the program has been executed successfully. If the label ERROR is encountered during the execution, the program contains a semantical error i.e. an error that has to do with the meaning of the program and not its form. In this case the program did not run successfully.

CHAPTER 2

OVERVIEW OF THE ALTRAN PROGRAMMING LANGUAGE

ALTRAN is a programming language designed mainly to enable the user of doing some symbolic manipulations on algebraics. (In ALTRAN we mean by algebraics rational fractions and in particular polynomials.) But naturally it allows other types of data like integer, rational, real, logical and label.

The basic arithmetic and logical operations are available and also some non-elementary operations namely substitution and function invocation.

The basic statements are the assignment statement, the goto statement and the if-group in addition to some other basic statements like end, return, continue etc. But other higher level statements or groups exist like the do-group and the procedure invocation.

A data language and I/O operations on this data language are available.

In what follows an overview of the language with some emphasis on certain features that seem important is given. A survey of the ALTRAN system can be found in [2] and the ALTRAN user's manual [1] gives a complete informal description of the language. This work is mainly based on this description. The modifications that we found convenient to make on the ALTRAN language (for the purpose of this work) are listed at the end of this chapter.

1. Constants and Variables

The integer constants, that can be short or long, do not lose their accuracy when manipulated among themselves; they are always exact. A rational constant is not basic, it can be considered as a pair of integer constants. When rational and integer constants or values are mixed with real constants or values they lose their accuracy (because of round off) and there is no way to recover from this loss.

Variables have in general seven attributes: type, structure precision, scope, storage class, language and value. But the value attribute can only be assigned to a dummy variable, i.e. a variable appearing as parameter in the definition of a procedure. The language attribute is given to a name of a procedure when it is specified in some other procedure. When the structure is array, it may be completed by a descriptor block (D.B.) giving the bounds of the array. If the D.B. is not specified the array bounds are dynamic and can be borrowed from other arrays during execution. When the type is algebraic, a layout giving the indeterminates and their maximum exponent may be specified. If not, the algebraic is dynamic and its layout is borrowed from another algebraic during execution.

Each variable should be declared by giving at least its type. This declaration can be made in several attribute statements and the variable may be initialized in only one of them.

```
e.g.  int  I, J;  alg (X:5, Y:5)  A = X+Y;
      external  A, I;
      long      J = 7.
```

2. Substitution

In an algebraic it is possible to substitute arithmetic expressions (with some restrictions) for the indeterminates. This can be done implicitly or explicitly. Also it can be of several levels.

e.g. If A is the previously declared algebraic,

A(1,2) is an implicit substitution that yields 3 as value (by replacing X by 1 and Y by 2).

A(1) yields 1+Y.

A(X,Y = 5,3) is an explicit substitution that yields the value 8.

A(Y,X)(1,2) is a 2 level substitution that yields the value 3.

3. Invocation

A function or a subroutine can be invoked by putting its name followed by a list of actual parameters that are in general expressions. When a procedure is invoked inside another procedure, it should be declared (or specified) by giving its name the language attribute (ALTRAN, FORTRAN or FOREIGN). If it is invoked as a function, the type as well as other attributes (when they are not the default attributes) should be specified. A matching of the actuals and dummies is made in the inbound transmission of arguments, i.e. assignment of each actual to the corresponding dummy. But it is not necessary to have as much actuals as there is dummies.

After execution of the body of the procedure an outbound transmission of the arguments is done i.e. assignment of each dummy to the corresponding actual if this is possible.

When it is a function invocation a return-statement returns the required value (arithmetic or logical). But when it is a subroutine then a return-statement can be used with a label argument to return to some point in the invoker.

4. Expressions

In an arithmetic expression a primary can be a constant, a scalar variable, an array variable, a substitution or a function invocation. However, a substitution is considered as an algebraic whatever it yields as value. Some rules for combining arrays as a whole between themselves or with scalars should be respected.

Logical expressions can be formed using some logical operators (or, and, eqv, neqv, ...).

A label expression is a label constant or a label variable.

5. Statements

Besides the basic statements, the invocation statement and the do-group, we have the I/O statements read and write (each followed by a list of arguments).

There is also the algebraic options statement which is an invocation of a built-in procedure. This procedure changes the global variables corresponding to the algebraic options that define the degree of

simplification to perform on any algebraic to compute. These options can be set at any time in a program by means of this statement. For more details about the algebraic options see [1].

6. Procedure

Typically, a main procedure has no arguments and is not recursive. Consequently, this is how we will view it in the formal description. A non-main procedure has a name and eventually, attributes and a list of dummies. These variables as well as the others used in the procedure should be declared in a list of attribute statements following the name and arguments of the procedure.

7. Program

A program consists of a main procedure and eventually, several non-main procedures. The execution starts at the beginning of the main procedure and ends with its end.

8. The Data Language

The input as well as the output make part of the ALTRAN system and constitutes the so-called data language. In this language we can have expressions in which the primaries are constants and indeterminates. An array notation for polynomials [1, B.4.6] and the back-reference [1, B.4.6] help making the presentation of the input more compact.

9. Restrictions on ALTRAN

The following restrictions or changes are made on the ALTRAN language as described in [1].

- Borrowing is not considered.
- Operations on arrays as a whole are excluded. Only elements of an array can be used in an expression. Consequently, there are no lists.
- Indeterminates can only be scalar.
- In the data language indeterminates are excluded and the alternative polynomial notation is not considered.
- A procedure name can only have the ALTRAN attribute, i.e. it can only be written in the ALTRAN language.

Some other minor modifications can be found in the description , e.g. not considering the relation operators .EQ., .NE., ... but only ==, < >, ... for the convenience of the presentation, distinguishing the keywords (lower case letters underlined), putting semi-colons between statements, etc. ...

10. Example

The following is a very simple example of an ALTRAN program. The program consists only of a main procedure and it computes and prints out the Chebyshev polynomials of degree less or equal to 5.

```
PROCEDURE MAIN
  ALGEBRAIC (X:5) R = 1, S = X,T
  INTEGER N
  WRITE "N = 0", R, "N = 1", S
  DO N = 2,5
     T = 2*X*S-R
     WRITE N, T
     R = S
     S = T
  DOEND
END
```

CHAPTER 3

FORMAL DESCRIPTION OF ALTRAN

Using the model for formal description based on the notion of linked forest manipulation system (l.f.m.s.) we describe in what follows the ALTRAN programming language with the restrictions stated in the previous chapter.

This description has two main parts: the Syntax Description part, and the Semantics Description part. Each of them is subdivided into several paragraphs corresponding to the different features of the language. Each of these paragraphs has two subdivisions and occasionally a third one:

Subdivision a) describes informally the way the formal part works and what it means. Subdivision b) is the formal part of the description. Subdivision c), when it exists, gives an example to clarify some ideas.

This formal description makes use of the following basic sets which are not completely specified but can be easily completed from the formal description if required.

$T_\ell = \{A,B,\ldots,Z\}$ is the *set of letter*

$T_s = \{+,-,\Theta,*,/,\uparrow,=,:,<,>,<=,>=,==,< >,\equiv,\not\equiv,\wedge,\vee,\neg,\ldots\}$

is the set of *special symbols*

T_w = {int, rat, proc,...} is the set of *word symbols*.

$T = T_\ell \cup T_s \cup T_w$ is the set of *terminals*.

N = {<int const>,<real const>, <stat>, <proc>,...} is the set of *non-terminals*.

C = {EXEC, LEXEC, VAL, LVAL, ...} is the set of control words.

F = {ineg, srt, intsh, cir, ...} is the set of basic functions.

$I_s = \{0,1,\ldots,L_s\}$ is the set of positive short integers.

$\bar{I}_s = \{0,-1,\ldots,-L_s\}$ is the set of negative short integers.

$I_\ell \quad \{L_s+1,\ldots,L_\ell\}$ is the set of positive long integers.

$\bar{I}_\ell = \{-(L_s+1),\ldots,-L_\ell\}$ is the set of negative long integers.

$Z = I_s \cup I_\ell \cup \bar{I}_s \cup \bar{I}_\ell$ is the set of signed integers.

R_s is the set of positive short real numbers.

R_ℓ is the set of positive long real numbers.

$R^+ = R_s \cup R_\ell$ is the set of positive real numbers.

R^- is the set of negative real numbers.

$R = R^+ \cup R^-$ is the set of real numbers.

$K = Z \cup R \cup$ {null} is the set of constants.

$\Sigma = T \cup K \cup C$ is the set of of input symbols.

$\Delta = T \cup K$ is the set of output symbols.

The formal description itself is a 5-tuple:

(Σ, Δ, F, SYNTAX, SEMANTICS)

where SYNTAX and SEMANTICS are described below:

SYNTAX = (Σ, N, A, B, F, J, R, <job>)

where - A is the set of label parameters with their respective domains.

The list of these label parameters and domains follows:

(λ , {A, B, ..., Z})

(δ, {0,1,...,9})

(χ_s, I_s), (χ_ℓ, I_ℓ), $(\chi, R \cup Z \cup I_e)$

$(\tau, \{\underline{int}, \underline{rat}, \underline{real}, \underline{log}, \underline{label}\})$, $(\tau_1, \{\underline{rat}, \underline{real}, \underline{log}, \underline{label}\})$

$(\beta, \{\underline{int}, \underline{rat}, \underline{indet}\})$, $(\beta_1, \{\underline{int}, \underline{rat}\})$

$(\pi, \{\underline{short}, \underline{long}\})$, $(\sigma, \{\underline{internal}, \underline{external}\})$

$(\gamma, \{\underline{autom}, \underline{static}\})$, $(\alpha, \{\underline{type}, \underline{prec}, \underline{st\ class}, \underline{struct}, \underline{scope},$
$\underline{lang}, \underline{value}\})$

$(\zeta, \{\underline{label}, \underline{dolab}\})$

$(\rho, \{<, <=, ==, < >, =>, > \})$, $(\rho_1, \{==, < >\})$

$(\mu, \{+, -, *, /, \uparrow, ==, < >, <, <=, >, >= \})$

$(\nu, \{+, -, *, /\})$

$(\phi \{101, 102, 103\})$, $(\psi, \{1, 2, 3\})$

$(\theta, \{INT, RAT, REAL, ALG\})$, $(\theta_1, \{INT, RAT, REAL\})$

$(\theta_2, \{RAT, REAL\})$, $(\theta_3, \{RAT, REAL, ALG\})$

$(\eta, \{ANY, INT, RAT\})$, $\{\eta, \{ANY, INT\})$.

- B is the set of tree parameters and their respective domains. The list of these tree parameters and their domains follows.

  $\{(t, \Sigma_*), (u, \Sigma_*), (v, \Sigma_*), (w, \Sigma_*),$
  $(x, \Sigma_*), (y, \Sigma_*), (z, \Sigma_*)\}$

- J is the set of production labels that can be retrieved from the formal description if required.

- R is the set of syntax rules given in subdivisions b) of the Syntax Description part.

SEMANTICS = (Σ, A', B', F, J', P)

where - A is the set of *label parameters* with their respective domains.

Their list follows:

(k,Z), (ℓ,Z), (m,Z), (n,Z), (p,Z^+)

$(\chi_1, Z \cup R)$, $(\chi_2, Z \cup R)$, $(\chi, Z \cup R)$

$(\tau, \{\underline{int}, \underline{rat}, \underline{real}, \underline{log}\})$, $(\tau_1, \{\underline{alg}, \underline{real}, \underline{int}, \underline{rat}\})$

$(\nu, \{+, -, *, /\})$

$(\eta, \{+, -, *, /\} \cup \{==, <>, >=, >, <=, <\})$

$(\mu, \{==, <>, >=, >, <=, <\})$

$(\xi, \{\underline{exp}, \underline{logexp}\})$, $(\tau, \{\underline{int}, \underline{rat}, \underline{real}, \underline{log}, \underline{label}\})$

$(\lambda, \{VAL, LVAL\})$, $(\pi, \{\underline{short}, \underline{long}\})$, $(\theta, \{INT, RAT, ALG\})$

$(\zeta, \{EXEC, LEXEC\})$, $(\alpha, \{\underline{var}, \underline{elem}\})$.

- B' is the set of *tree parameters* with their respective domains.

  Their list follows:

  $\{(t,\Sigma_*)$, (u,Σ_*), (v,Σ_*), (w,Σ_*),

  (x,Σ_*), (y,Σ_*), (z,Σ_*), (x_1,Σ_*), $(x_2,\Sigma_*)\}$

- J' is the set of *production labels* that can be relieved from the formal description.

- P is the set of *semantics productions* given in subdivisions b) of the Semantics Description part.

I. The Syntax Description

1. Basic Tokens

1.1 Letters and digits

a) The set of letters is $T_\ell = \{A,B,\ldots,Z\}$, and the set of digits is $T_d = \{0,1,\ldots,9\}$. λ and δ are label parameters, their respective domains being T_ℓ and T_d.

b)

<letter> → λ 1 $\lambda \in \{A,B,\ldots,Z\}$	$\circ\lambda$
<digit> → δ 2 $\delta \in \{0,1,\ldots,9\}$	$\circ\delta$

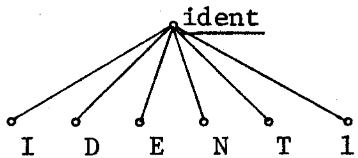
1.2 Identifiers

a) An identifier should start with a letter which can be eventually followed by letters and/or digits.

b)

<ident> → <letter> 1	<u>ident</u> <letter>
<ident> → <ident> <letter> 2	<ident> <letter>
<ident> → <ident><digit> 3	<ident> <digit>

c) The identifier IDENT1 is represented by (or generates) the following subtree



1.3 Arithmetic Constants

a) There are two types of arithmetic constants, the integer constants and the real constants.

b)

<arith const> → <int const>	<int const>
<arith const> → <real const>	<real const>

1.3.1 Integer constants

a) Depending on its value, an integer constant may be considered as a short integer constant (member of $I_s = \{0,1,\dots,L_s\}$) or as a long integer constant (member of $I_\ell = \{L_s+1, L_s+2, \dots, L_\ell\}$). The two limits L_s and L_ℓ are implementation dependent. χ_s and χ_ℓ are label parameters, I_s and I_ℓ being their respective domains.

b)

<int const> → <sh int const> 1	° <sh int const>
<int const> → <lg int const> 2	° <lg int const>
<sh int const> → χ_s 3 $\chi_s \in I_s$	int, short const, χ_s
<lg int const> → χ_ℓ 4 χ_ℓ I_ℓ	int, long const, χ_ℓ

$I_s = \{0,1,\ldots,L_s\}$, $I = \{L_s+1, L_s+2, \ldots, L_\ell\}$.

1.3.2 Signed integer constants

a) A signed integer constant is an integer constant preceded eventually by a sign (+ or -). When it is -, the constant is replaced by its negative value from either $\bar{I}_s$ or $\bar{I}_\ell$, using an l.f.m.s. and a basic function for negative.

b)

<s int const> → <int const> 1	° <int const>
<s int const> → + <int const> 2	° <int const>
<s int const> → -<int const> 3	⊖ <int const>

START	⊖ int, π const, χ	→	int, π const, ineg(χ)	STOP	

$\bar{I}_s = \{0,-1,\ldots,-L_s\}$, $\bar{I}_\ell = \{-(L_s+1),-(L_s+2),\ldots,-L_\ell\}$

$Z = I_\ell \cup I_s \cup \bar{I}_\ell \cup \bar{I}_s$

ineg: $Z \to Z$

$z \to -z$

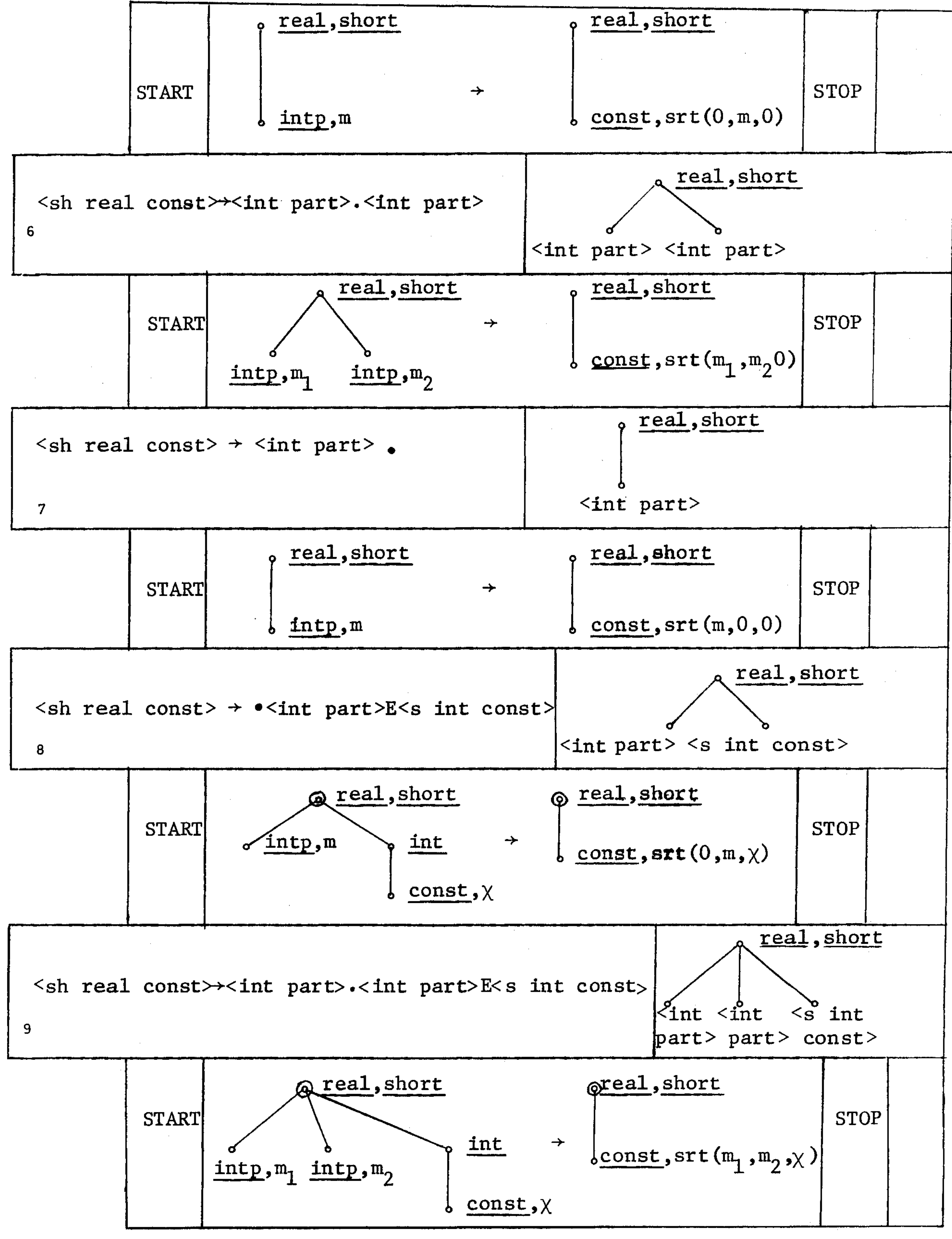
1.3.3 Real constants

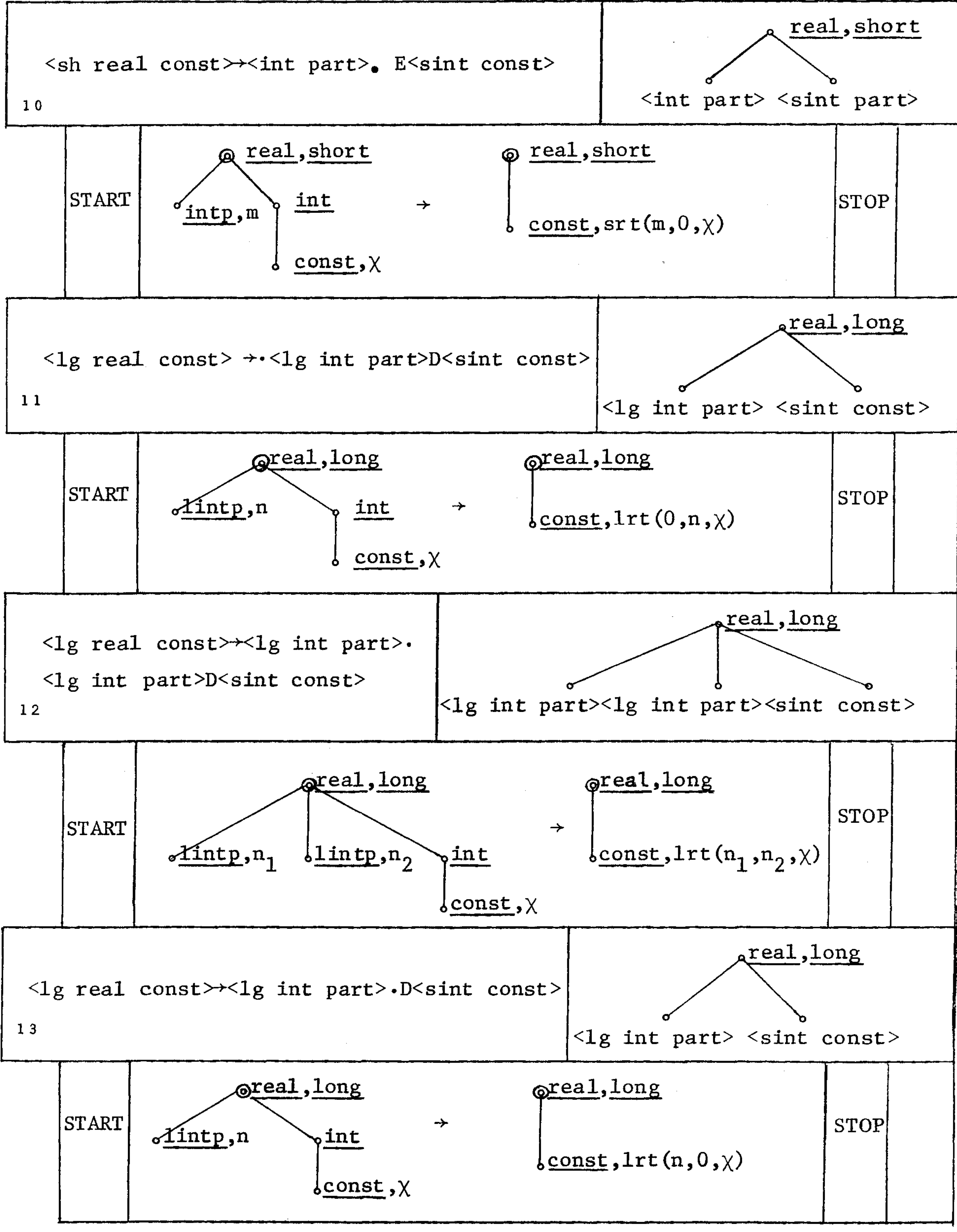
a) A real constant is considered to be short or long depending on its form rather than its value. It is the "D" between the mantissa and the exponent that will force the constant to be long.

For each possible representation of a real constant we use an l.f.m.s. to transform it to a real constant. For this purpose two basic functions are used, one for short real transformation (srt) and the other for long real transformation (lrt).

b)

<real const> → <sh real const> 1	° <sh real const>
<real const> → <lg real const> 2	° <lg real const>
<int part> → m m ∈ {0,1,...,p} 3	° intp,m
<lg int part> → n n ∈ {0,1,...,9} 4	° lintp,n
<sh real const> → .<int part> 5	° real,short | ° <int part>


START
real,short
intp,m
→
real,short
const,srt(0,m,0)
STOP
<sh real const>→<int part>.<int part>
6
real,short
<int part> <int part>
START
real,short
intp,m1 intp,m2
→
real,short
const,srt(m1,m2,0)
STOP
<sh real const> → <int part> •
7
real,short
<int part>
START
real,short
intp,m
→
real,short
const,srt(m,0,0)
STOP
<sh real const> → •<int part>E<s int const>
8
real,short
<int part> <s int const>
START
real,short
intp,m
int
const,χ
→
real,short
const,srt(0,m,χ)
STOP
<sh real const>→<int part>.<int part>E<s int const>
9
real,short
<int part> <int part> <s int const>
START
real,short
intp,m1 intp,m2
int
const,χ
→
real,short
const,srt(m1,m2,χ)
STOP



<sh real const>→<int part>. E<sint const>
10
real,short
<int part> <sint part>
START
real,short
intp,m
int
const,χ
→
real,short
const,srt(m,0,χ)
STOP
<lg real const> →·<lg int part>D<sint const>
11
real,long
<lg int part> <sint const>
START
real,long
lintp,n
int
const,χ
→
real,long
const,lrt(0,n,χ)
STOP
<lg real const>→<lg int part>·
<lg int part>D<sint const>
12
real,long
<lg int part><lg int part><sint const>
START
real,long
lintp,n1
lintp,n2
int
const,χ
→
real,long
const,lrt(n1,n2,χ)
STOP
<lg real const>→<lg int part>·D<sint const>
13
real,long
<lg int part> <sint const>
START
real,long
lintp,n
int
const,χ
→
real,long
const,lrt(n,0,χ)
STOP

$I_p = \{0,1,\ldots,p\}$ $I_q = \{0,1,\ldots,q\}$ $q > p$

$I_e = \{0,1,\ldots,e\}$

$srt : I_p \times I_p \times I_e \to R_s$

$lrt : I_q \times I_q \times I_e \to R_\ell$

R_s and R_ℓ are subsets of the set of real numbers with $R_s \subset R_\ell$.

$srt(m_1,m_2,n)$ is defined iff $\overline{m_1 m_2} \in I_p$.

$lrt(m_1,m_2,n)$ is defined iff $\overline{m_1 m_2} \in I_q$.

1.4 Logical constants

a) There are two logical constants true and false.

b)

<log const> → true	log | const,true
<log const> → false	log | const,false

2. Attribute statements

2.1 Attributes

a) There are seven different attributes for an identifier.

- The type can be integer, rational, algebraic, real, logical or label. The layout for algebraics makes part of the type and it consists of pairs, indeterminate: highest exponent. An indeterminate is an identifier and its highest exponent is an integer expression, i.e. an expression in which all constants are integer constants and all variables are integer variables.

  The occurrence of an indeterminate in a layout is an implicit declaration of this indeterminate. Thus no other declaration of this identifier or occurrence in another layout is permitted (as we shall see later on).
- The structure can be scalar or array followed by a descriptor block (D.B.) that gives the lower and upper bounds for each dimension. The lower bound might be implicit, in which case it is 1.
- The precision can be either short or long.
- The scope can be either internal or external.
- The storage class can be either automatic or static.
- The language attribute can be only altran (for the purpose of this description).
- The value attribute can only be value.

b)

No.	Rule	Tree
1	<indet> → <ident>	indet,dcl <ident>
2	<layout> → <indet>:<int exp>	layout <indet> <int exp>
3	<layout> → <layout>,<indet>:<int exp>	<layout> <indet> <int exp>
4	<type> → τ $\tau \in$ {int,rat,real,log,label}	type τ
5	<type> → alg(<layout>)	type alg <layout>
6	<D.B.> → <int exp>	int,short <int exp> const,1
7	<D.B.> → <int exp>:<int exp>	<int exp> <int exp>
8	<D.B.> → <D.B.>,<int exp>	<D.B.> int,short <int exp> const,1

	Rule	Tree
9.	<D.B.> → <D.B.>,<int exp>:<int exp>	<D.B.> <int exp> <int exp>
10	<struct> → scal	struct scal
11	<struct> → array (<D.B.>)	struct array <D.B.>
12	<precision> → π $\pi \in$ {short,long}	prec π
13	<scope> → σ $\sigma \in$ {internal,external}	scope σ
14	<st class> → γ $\gamma \in$ {autom,static}	st class γ
15	<lang> → altran	lang altran
16	<value> → value	val value
17	<attrib> → <type >	<type>
18	<attrib> → <struct>	<struct>

<attrib> → <precision> 19	° <precision>
<attrib> → <scope> 20	° <scope>
<attrib> → <st class> 21	° <st class>
<attrib> → <lang> 22	° <lang>
<attrib> → <value> 23	° <value>

2.2 The attribute statement

a) An attribute statement consists of a list of attributes followed by a list of subjects. A subject is either an identifier or an initialization of an identifier. An l.f.m.s. is used to check that no identifier and no attribute appears more than once in an attribute statement.

b)

<ℓ.attrib> → <attrib> 24	ℓ.attrib / <attrib>
<ℓ.attrib> → <ℓ.attrib><attrib> 25	<ℓ.attrib> <attrib>
<subject> → <ident> 26	<ident>

Rule	Tree
<subject> → <ident> = <exp> 27	init <ident> =,stat var exp used <exp> <ident>
<subject> → <ident> = <log exp> 28	init <ident> =,stat var log exp used <log exp> <ident>
<subject> → <ident> = <label exp> 29	init <ident> =,stat var used <label exp> <ident>
<ℓ.subjects> → <subject> 30	ℓ.subj <subject>
<ℓ.subjects>→<ℓ.subjects>,<subject> 31	<ℓ.subjects> <subject>
<attrib stat> → <ℓ.attrib><ℓ.subjects> 32	decl <ℓ.subjects> <ℓ.attrib>

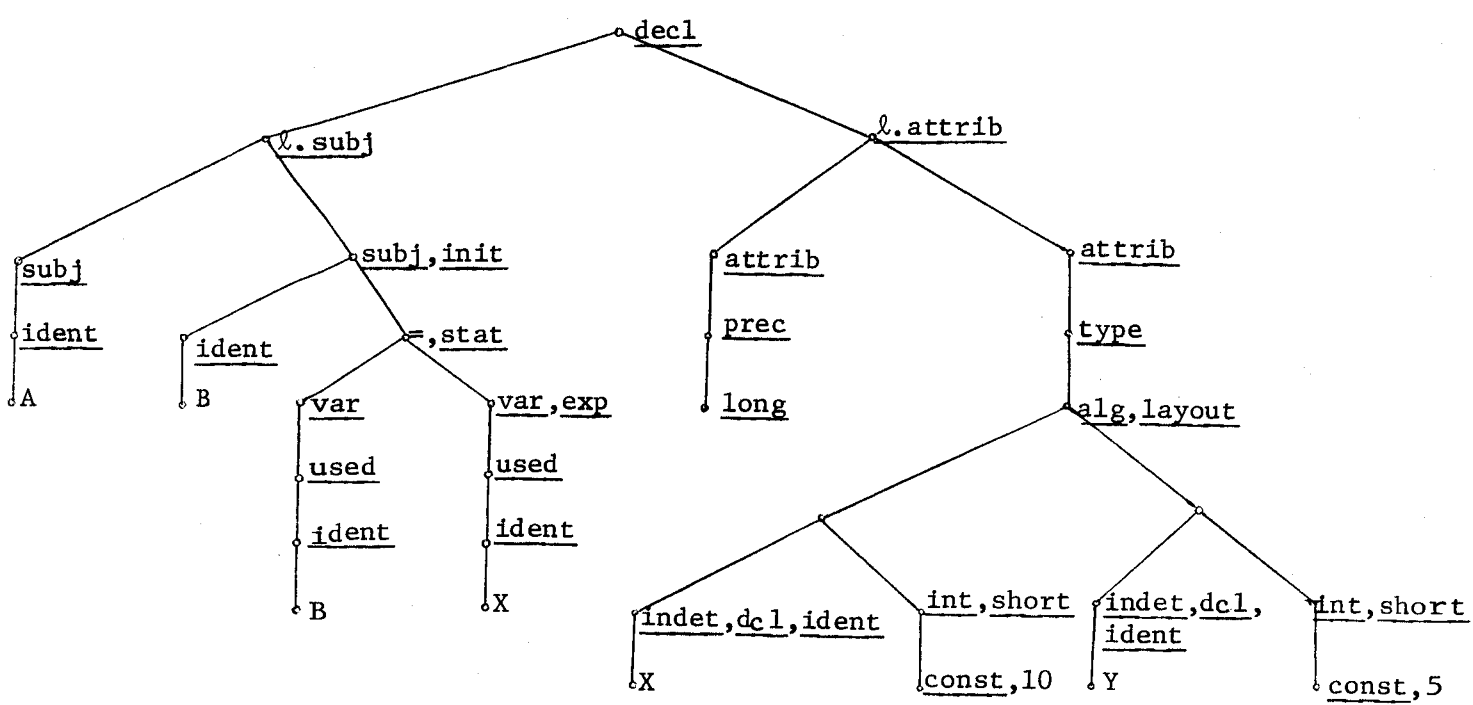
START	ℓ.subj; ident u, ident u →	ERROR	E_1
E_1	decl; ℓ; α, α →	ERROR	STOP

$\alpha \in$ {type, struct, prec, scope, st class, lang, value}.

c) The attribute statement:

long alg (X:10, Y:5) A,B = X

would be represented by the following tree:



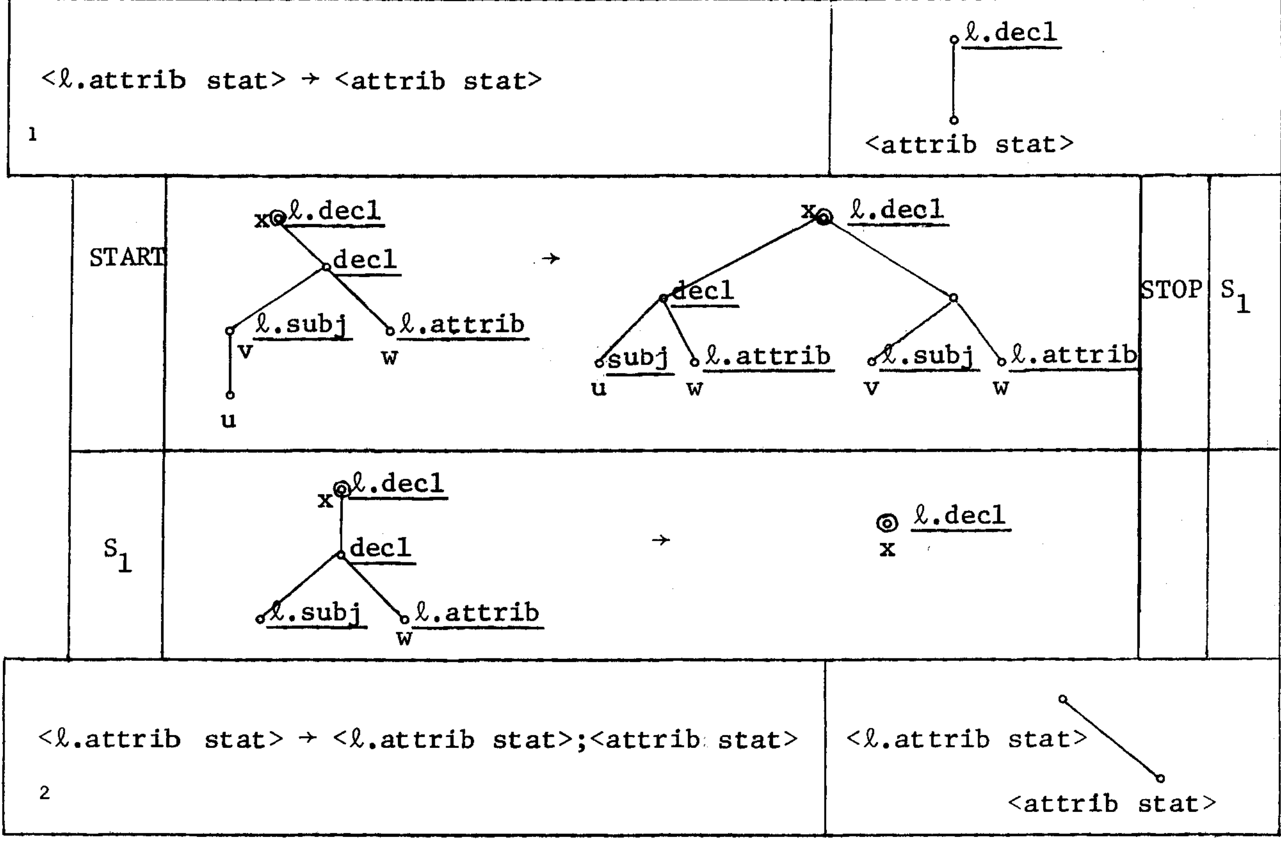
2.3 List of attribute statements

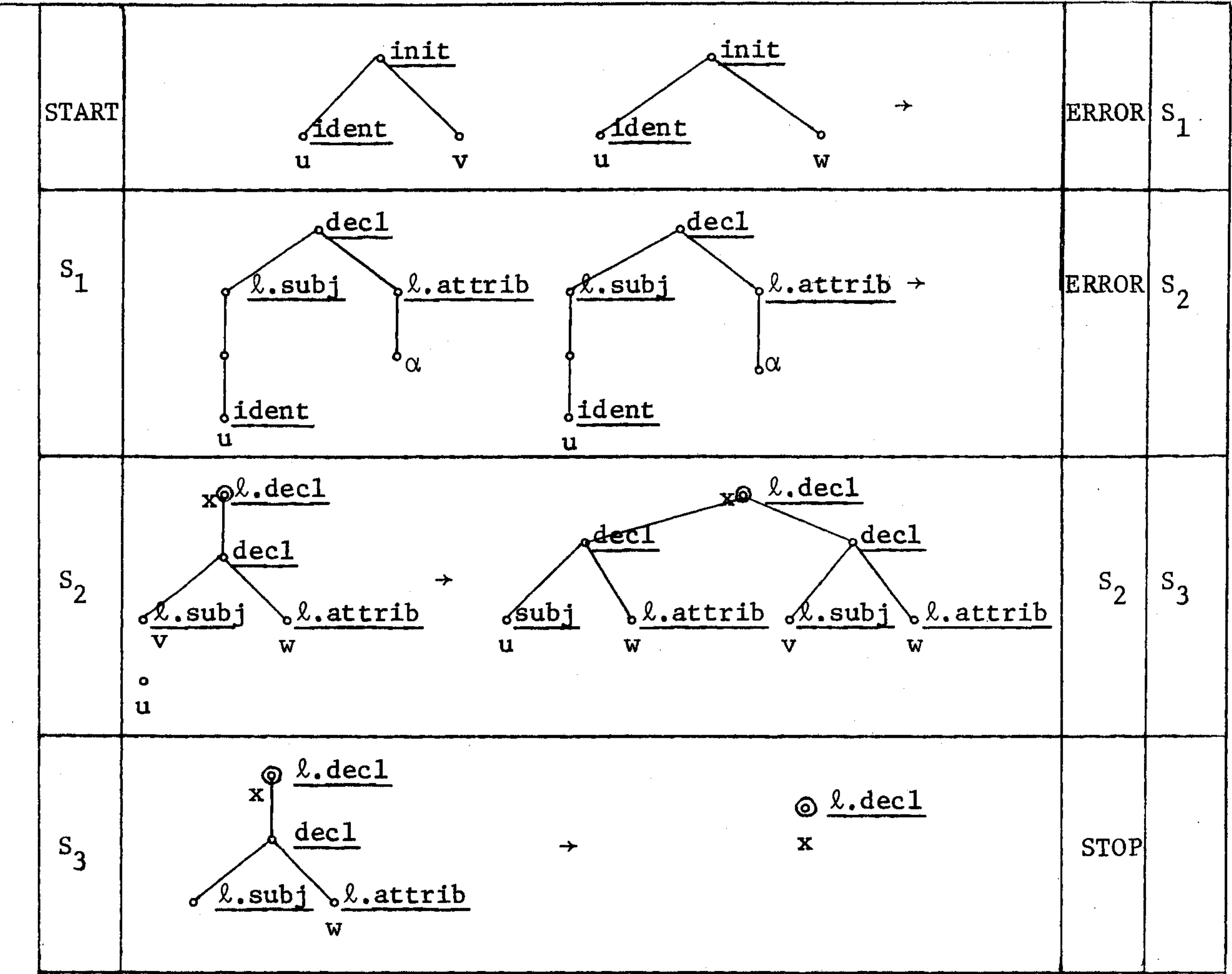
a) A list of attribute statements consists of several attribute statements separated by semi-colons.

While an identifier can be assigned attributes in several attribute statements, the same attribute should not be assigned more than once ($2.S_1$). Also an identifier should not be initialized more than once in one or more attribute statements (2.START).

After this checking, each attribute statement generates as many attribute statements as there is subjects. The order in which the subjects appear is respected ($2.S_2$). This is repeated for each list of attribute statements that is constructed.

b)





3. Arithmetic, logical and label expressions

3.1 Arithmetic expressions

a) An arithmetic expression is constructed using the elementary operations +, -, *, /, ↑ with the conventional priorities. However, it is not allowed to write A↑B↑C; one should write either (A↑B)↑C or A↑(B↑C). The nonterminal <simple factor> is introduced to avoid such a construction.

Subscripted variables are allowed and can be used as a primary in an expression. The subscripts should be integer expressions. The operation

of getting the corresponding element of an array is called element accessing; that is why the tree corresponding to a subscripted variable is labeled elem.

A function invocation is considered as a primary and can be used accordingly.

A substitution is not a primary for it cannot appear as exponent in any case. A substitution can be explicit (8) or implicit (6,7). In the first case the indeterminates or more generally the forms (variables having indeterminates as values) to be substituted are indicated. In the second case they are not, but the expressions to substitute for are in the same order as the indeterminates in the layout.

For the explicit substitution, that can be at several levels, the forms are paired with their corresponding expressions using an l.f.m.s. that consists of two productions.

b)

Production	Tree
<var> → <ident> 1	var used <ident>
<var> → <ident>(<ℓ.int exp>) 2	elem var <ℓ.int exp> used <ident>
<form> → <var> 3	form <var>

	Rule	Tree
4	<ℓ.forms> → <form>	ℓ.forms <form>
5	<ℓ.forms> → <ℓ.forms>,<form>	<ℓ.forms> <form>
6	<subst> → <exp>(<ℓ.exp>)	subst exp <exp> <ℓ.exp>
8	<subst> → <exp>(<ℓ.forms> = <ℓ.exp>)	subst exp < exp> <ℓ.forms> <ℓ.exp>

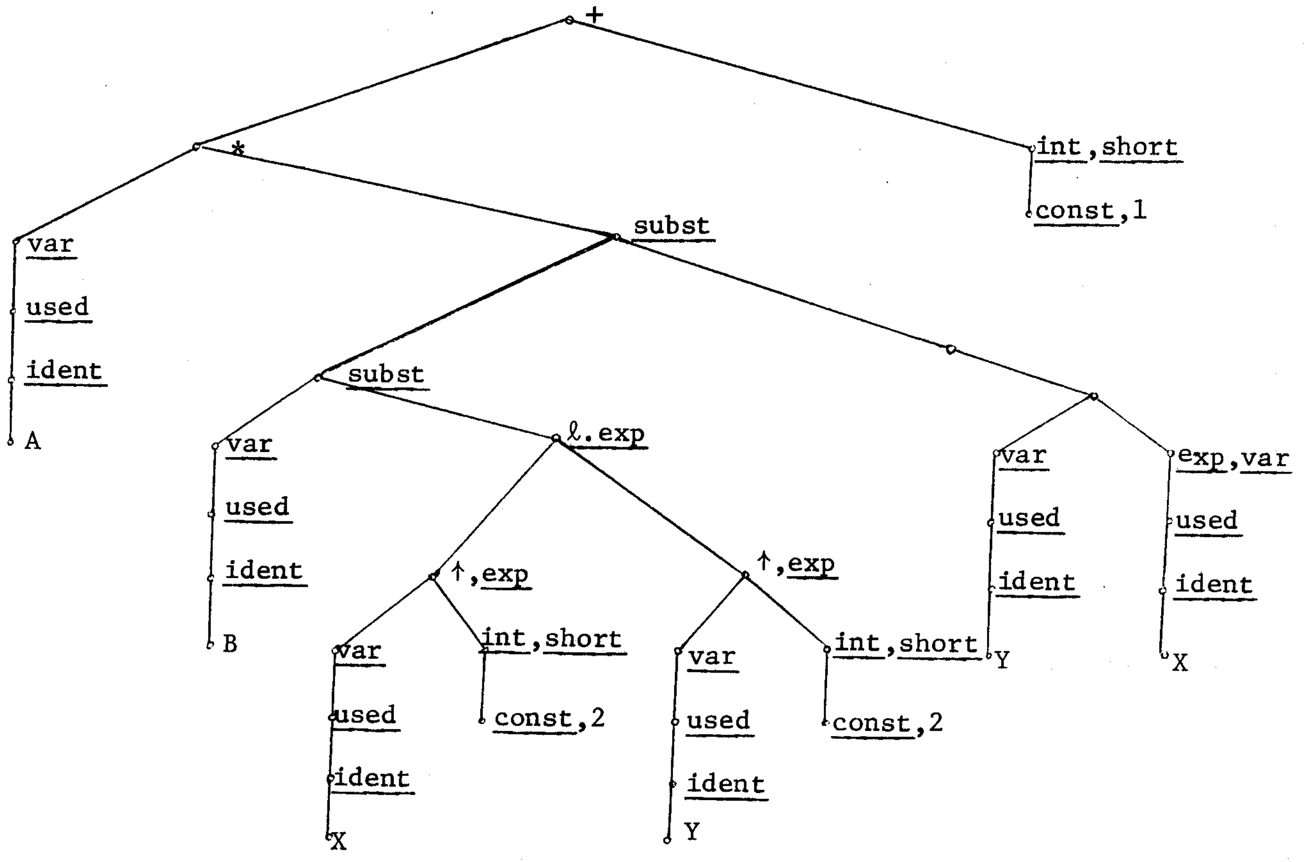
START	w ℓ.forms x ℓ.exp y u v	→	w ℓ.forms x ℓ.exp y u v	START	L_1
L_1	w ℓ.forms ℓ.exp	→	w	STOP	ERROR

10	<prim> → <const>	∘ <const>
11	<prim> → <var>	∘ <var>
12	<prim> → (<exp>)	∘ <exp>
13	<prim> → <invocation>	∘ <u>fnct</u> <invocation>
14	<simple factor> → <prim>	∘ <prim>
15	<simple factor> → <prim>**<prim>	↑ <prim> <prim>
16	<simple factor> →<subst>	∘ <subst>
17	<factor> → <simple factor>	∘ <simple factor>
18	<factor> → (<factor>)**<prim>	↑ <factor> <prim>

No.	Rule	Tree
19	<term> → <factor>	<factor>
20	<term> → <term>*<factor>	* <term> <factor>
21	<term> → <term>/<factor>	/ <term> <factor>
22	<exp> → <term >	<term>
23	<exp> → +<term>	<term>
24	<exp> → -<term>	⊖ <term>
25	<exp> → <exp> +<term>	+ <exp> <term>
26	<exp> → <exp>-<term>	- <exp> <term>
27	<ℓ.exp> → <exp>	ℓ.exp exp <exp>
28	<ℓ.exp> → <ℓ.exp>,<exp>	<ℓ.exp> exp <exp>

Rule	Tree
<int exp> → <exp> 29	exp,int <exp>
<ℓ.int exp> → <int exp> 30	ℓ.exp <int exp>
<ℓ.int exp> → <ℓ.int exp>,<int exp> 31	<ℓ.int exp> <int exp>

c) The expression A*B(X**2, Y**2) (Y = X)+1 is represented by:



3.2 Logical expressions

a) A logical expression is formed using logical constants, variables (including element accessing from an array), relations function invocations and logical operators which are ¬, ∧, ∨, ≡, ≢.

A relation can be formed using the operators <, <=, ==, <>, >=, >, to compare two arithmetic expressions. But for arithmetic expressions with algebraic operands only == and <> are defined (as we will see in the semantics description).

b)

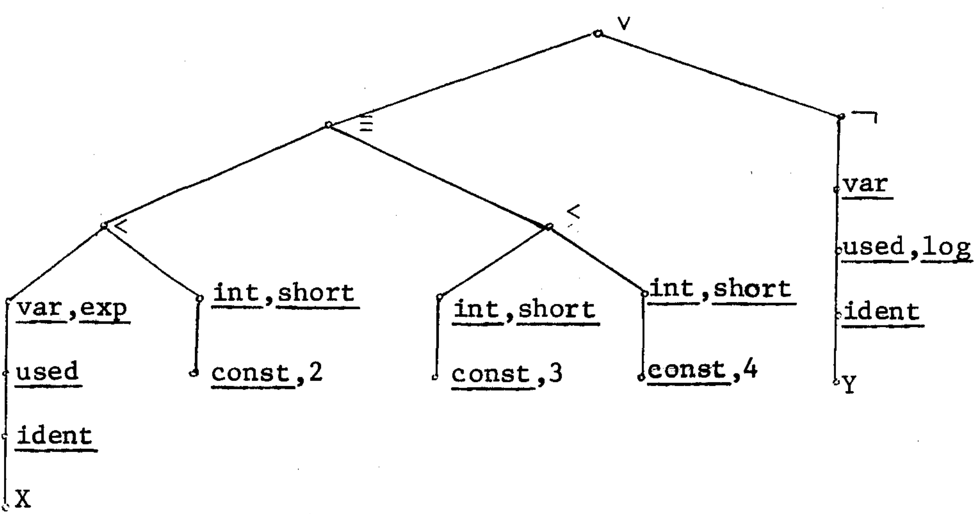
No.	Rule	Tree
1	<log var> → <ident>	var used,log <ident>
2	<log var> → <ident> (<ℓ.exp>)	elem var <ℓ.exp> used,log <ident>
3	<relation> →<exp>ρ<exp> ρ ∈ {<,<=,==,<>,>=,>}	ρ exp exp <exp> <exp>
4	<log prim> → <log const>	<log const>
5	<log prim> → <log var>	<log var>
6	<log prim> → <invocation>	<invocation>
6'	<log prim> →(<log exp>)	<log exp>

	Production	Tree
7	<log prim> → <relation>	<relation>
8	<log factor> → <log prim>	<log prim>
9	<log factor> → not<log prim>	¬ <log prim>
10	<log term> → <log factor>	<log factor>
11	<log term> → <log term> and <log factor>	∧ <log term> <log factor>
12	<log super term> → <log term>	<log term>
13	<log super term> → <log super term>or<log term>	∨ <log super term><log term>
14	<log exp> → <log super term>	<log super term>
15	<log exp>→<log super term>eqv<log super term>	≡ <log super term><log super term>
16	<log exp>→<log super term>neqv<log super term>	≢ <log super term><log super term>

c) The logical expression:

(X < 2 eqv 3 < 4) or not Y

would be represented by:



3.3 Label expressions

a) A label expression can be a label constant (label) or a label variable. The latter can be an identifier or an element of an array.

Note that a function invocation cannot give a label as value.

b)

<label> → <ident> 1	used,label │ <ident>
<label var> → <ident> 2	var used,label │ <ident>

<label var> → <ident>(<ℓ.int exp>) 3	elem / var / used,label / <ident> ; <ℓ.int exp>
<label exp> → <label> 4	<label>
<label exp> → <label var> 5	<label var>

4. Statements

4.1 Elementary statements

a) 4.1.1 The assignment statement: It is only possible to assign an arithmetic expression to an arithmetic variable, a logical expression to a logical variable and a label expression to a label variable.

4.1.2 The return-statement: The return-statement can have several forms. The simplest is a return with no arguments. But there is also a return with an arithmetic or logical expression as argument, to be eventually returned as value of a function. There is also the return with a label variable as argument which should be a dummy variable in a subroutine procedure, and it gives the return point in the invoker.

4.1.3 The goto-statement: It is a goto followed by a label expression.

4.1.4 The continue, doend, and end statements:

4.1.5 The Input/Output statements: It can be either a write followed by a list of arguments or a read followed by a list of arguments.

b)

<stat> → <var> = <exp> 1	=,stat <var> <exp>
<stat> → <log var> = <log exp> 2	=,stat <log var> <log exp>
<stat> → <label var> = <label exp> 3	=,stat <label var> <label exp>
<stat> → return 4	stat return
<stat> → return (<exp>) 5	stat return,exp <exp>
<stat> → return (<log exp>) 6	stat return,logexp <log exp>
<stat> → return <dummy label> 7	stat return <dummy label>
<dummy label> → <label var> 8	dum lab <label var>
<stat> → goto <label exp > 9	goto,stat <label exp>

No.	Production	Tree
10	<stat> → continue	continue,stat
11	<end-stat> → end	end,stat
12	<end stat> → <ident>:<end-stat>	<end-stat> label,dcl <ident>
13	<doend-stat> → doend	doend,stat
14	<doend-stat> → <ident>:<doend-stat>	<doend-stat> label,dcl <ident>
15	<char string> → ξ $\xi \in T_\ell \cup T_d \cup T_s$.	string ξ
16	<char string> → <char string> ξ	<char string> ξ
17	<I/O arg> → "<char string>"	arg <char string>
18	<I/O arg> → <var>	arg <var>
19	<I/O arg> → <log var>	arg <log var>
20	<ℓ.I/O arg> → <I/O arg>	ℓ.arg <I/O arg>
21	<ℓ.I/O args> → <ℓ.I/O args>,<I/O arg>	<ℓ.I/O args> <I/O arg>

Rule	Tree
<stat> → read<ℓ.var> 22	∘ read,stat <ℓ.var>
<stat> → write <ℓ.I/O args> 23	∘ write,stat <ℓ.I/O args>
<ℓ.var> → <var> 24	ℓ.var <var>
<ℓ.var> → <log var> 25	ℓ.var <log var>
<ℓ.var> → <ℓ.var>,<var> 26	<ℓ.var> <var>
<ℓ.var> → <ℓ.var>, <log var> 27	<ℓ.var> <log var>

4.2 Invocation statements

a) 4.2.1 Dummy and Actual Variables: The dummies are variables of any type, while the actuals are expressions of any type but they can be omitted (blank) explicitly or implicitly. It is also possible that an actual be an array variable.

4.2.2 Invocation Statement: It is an invocation of a subroutine procedure with or without parameters. The number of actuals do not need be the same as that of the dummies.

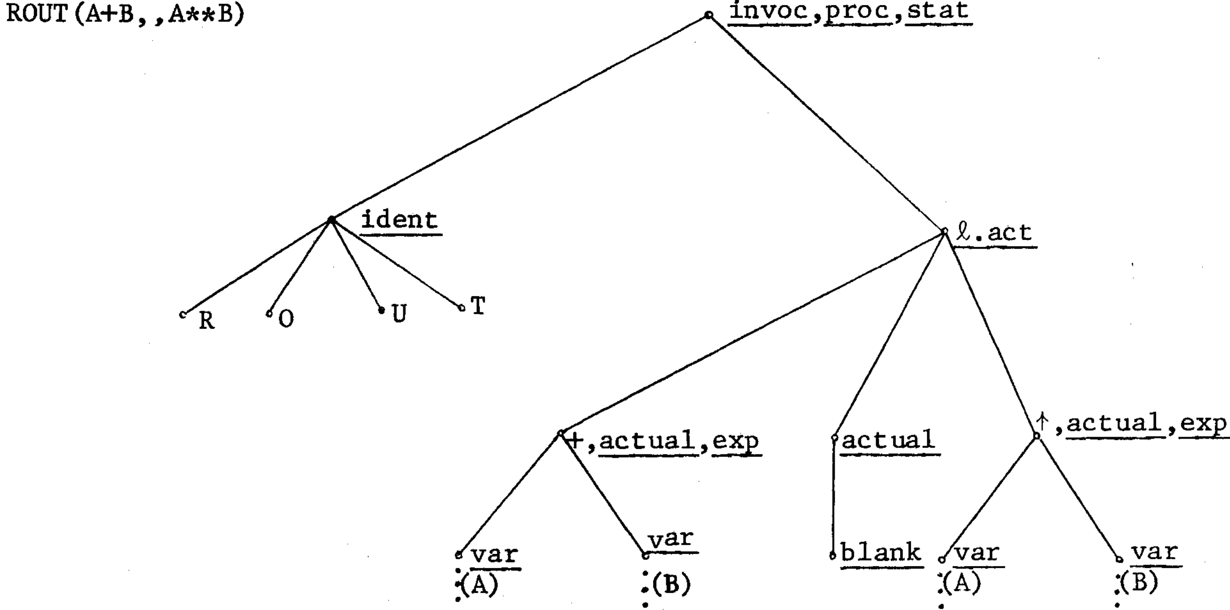
4.2.3 Algebraic Options Statement: This statement is in fact an invocation of a procedure (built-in) OPTS having two arguments. This procedure sets the required algebraic options concerning the simplification of algebraics.

b)

No.	Production	Tree
1	<dum> → <var>	dum,var used <ident>
2	<ℓ.dum> → <dum >	ℓ.dum <dum>
3	<ℓ.dum> → <ℓ.dum>,<dum>	<ℓ.dum> <dum>
4	<actual> → ε	actual blank
5	<actual> → <exp>	actual,exp <exp>
6	<actual> → <log exp>	logexp <log exp>
7	<actual> → <label exp>	<label exp>
8	<actual> → <array var>	actual <array var>
9	<array var> → <var>	array <var>
10	<ℓ.act> → <actual>	ℓ.act <actual>
11	<ℓ.act> → <ℓ.act>,<actual>	<ℓ.act> <actual>
12	<invocation> → <ident>(<ℓ.act>)	invoc <ident> <ℓ.act>

<stat> → <invocation> 13	∘ proc,stat <invocation>
<stat> → <ident> 14	invoc,proc,stat <ident> ℓ.act actual blank
<stat> → opts(ϕ,ψ) $\phi \in \{101,102,103\}$, $\psi \in \{1,2,3\}$ 15	opts ϕ ψ

c) The following procedure invocation where A and B are arithmetic variables has 3 actuals, one of them is explicitly omitted and it has the following tree representation.



5. Groups

a) 5.1 The do-group

A do-group consists of a body (list of statements) between a do-statement and a doend-statement.

The do-group tree is replaced, using an l.f.m.s., by a list of basic statements and the if-group which reflects its meaning (D_1).

To avoid a jump from the outside into a do-group (which is forbidden), the usage is linked to the declaration for each label and in the declaration tree, label is replaced by dolab (D_2,D_3).

5.2 The if-group

An if-group can be followed eventually by an else-group. But the else-group cannot appear by itself.

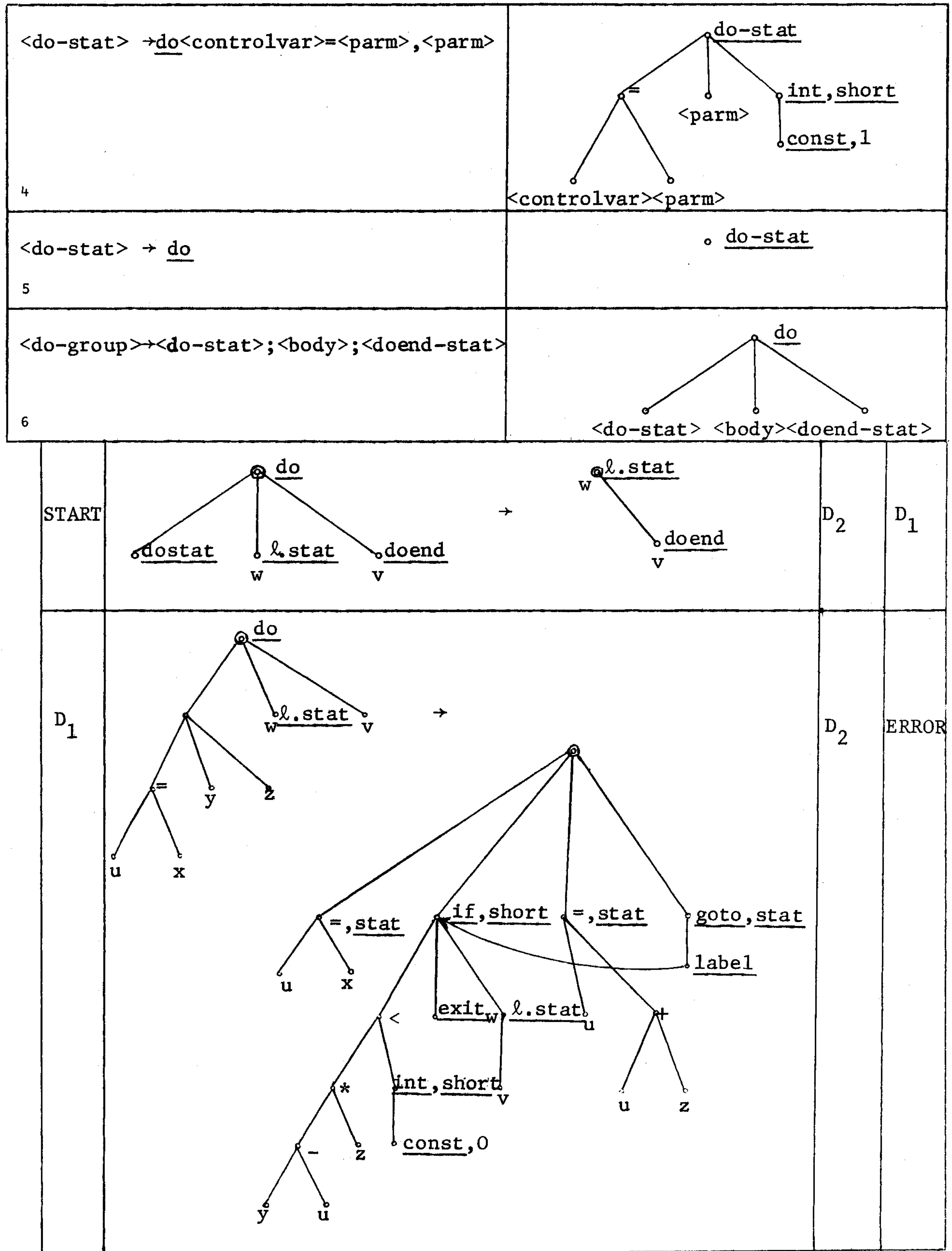
5.3 Labeled and unlabeled group - Body

A group of statements may be labeled, but when it makes part of an if-group then it should be unlabeled.

A Body is a list of groups separated by semi-colons.

b)

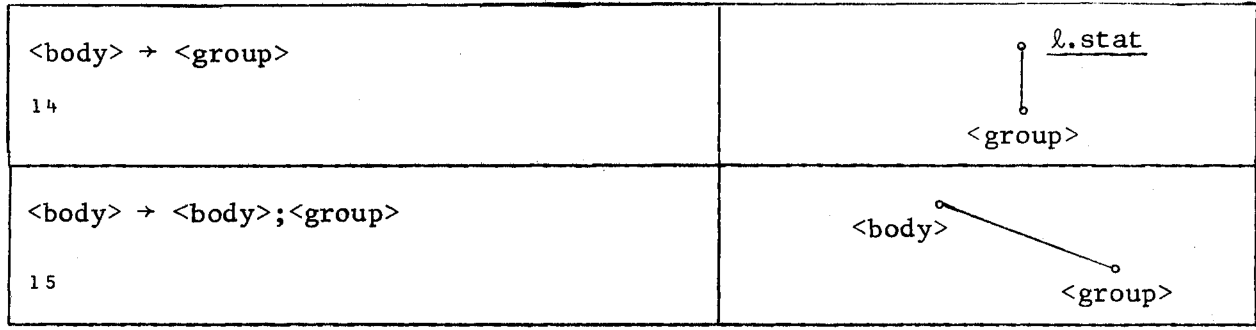
No.	Rule	Tree
1	<parm> → <int exp>	<int exp>
2	<control var> → <ident>	var used,int,short <ident>
3	<do-stat> → do<control var>=<parm>,<parm>,<parm>	dostat = <parm><parm> <controlvar><parm>



<do-stat> →do<controlvar>=<parm>,<parm>
4
do-stat
=
<parm>
int,short
const,1
<controlvar><parm>
<do-stat> → do
5
do-stat
<do-group>→<do-stat>;<body>;<doend-stat>
6
do
<do-stat> <body><doend-stat>
START
do
dostat
ℓ.stat
doend
w
v
→
ℓ.stat
w
doend
v
D2
D1
D1
do
ℓ.stat
w
v
=
y
z
u
x
→
=,stat
u
x
if,short
=,stat
goto,stat
label
<
exit
ℓ.stat
w
u
+
*
int,short
v
u
z
-
z
const,0
y
u
D2
ERROR

D_2	u ζ,dcl goto / used,label / ident u → dolab,dcl goto / label $\zeta \in$ {label,dolab}	D_2	D_3
D_3	u ζ,dcl = / used,label / ident u → dolab,dcl u = / label $\zeta \in$ {label,dolab}	D_3	STOP

<if-group> →if(<log exp>)<unlabeled group> 7	if,stat <log exp> <unlabeled group> end,stat
<if-group>→if(<log exp>)<unlabeled group>; else<unlabeled group> 8	if,stat <log exp> <unlabeled group> <unlabeled group>
<unlabeled group> → <if group> 9	<if-group>
<unlabeled group> → <do-group> 10	<do-group>
<unlabeled group> → <stat> 11	<stat>
<group> → <unlabeled group> 12	<unlabeled group>
<group> → <ident>:<group> 13	label,dcl <ident>



<body> → <group> 14	ℓ.stat <group>
<body> → <body>;<group> 15	<body> <group>

ζ ∈ {label,dolab}.

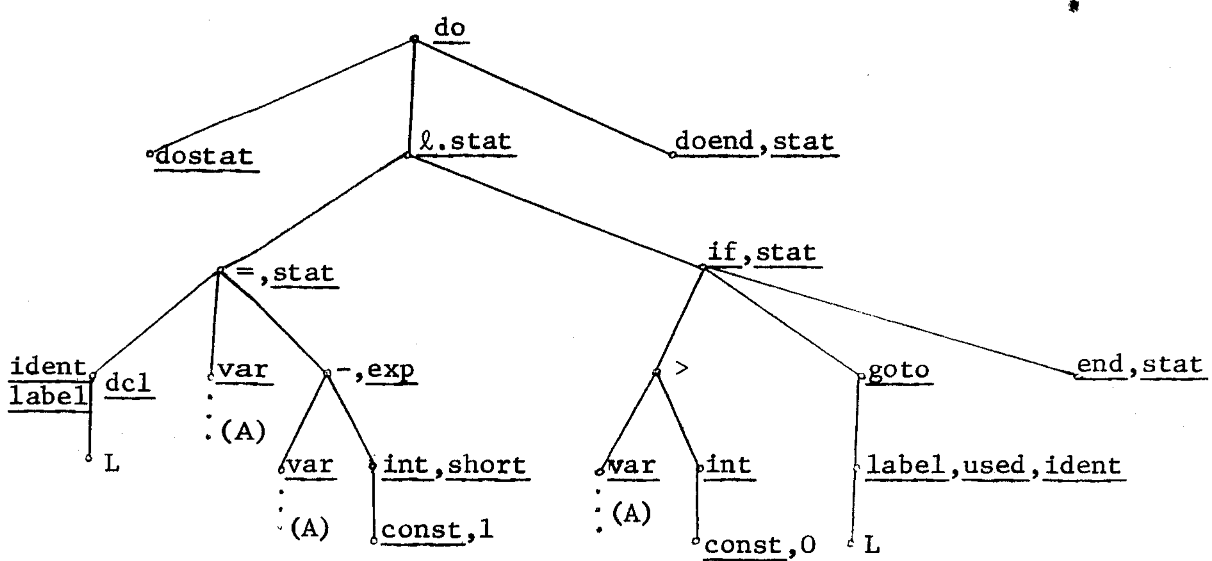
c) Consider the following do-group:

do; L : A = A-1;

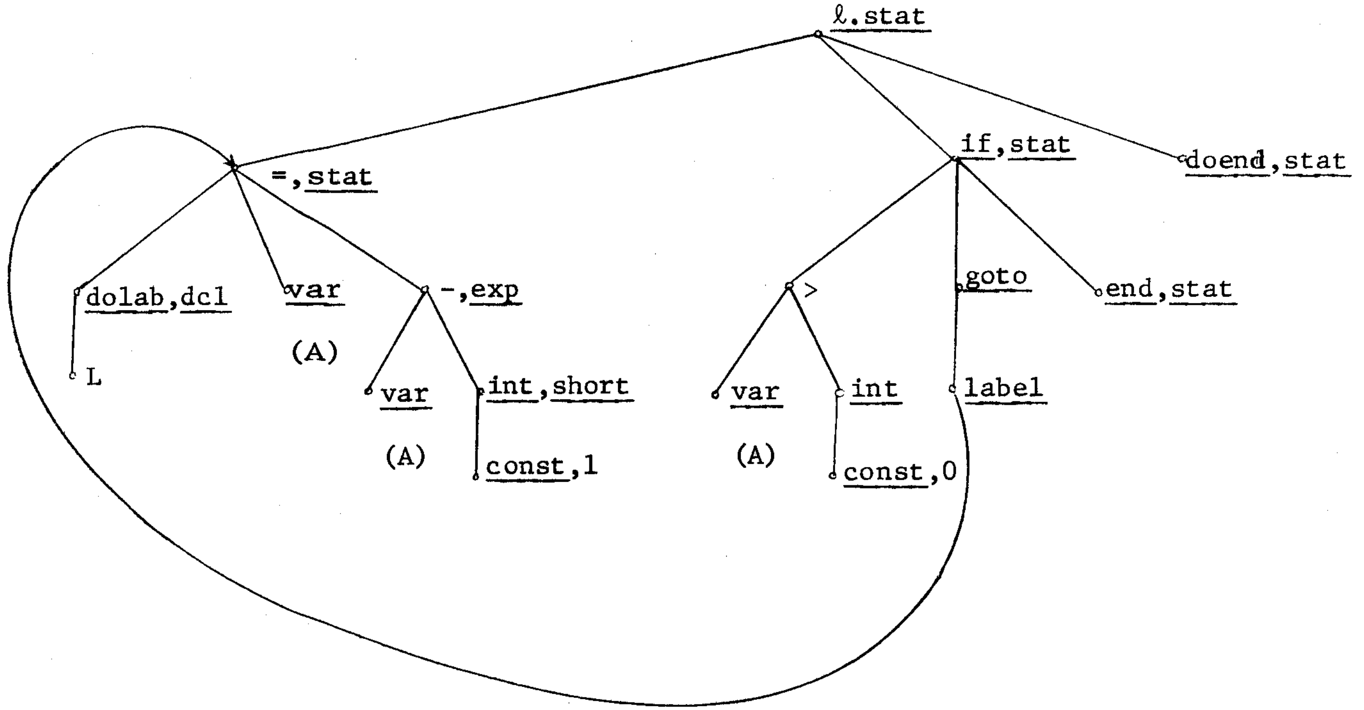
if (A > 0) goto L;

doend

Before any transformation using the l.f.m.s. is performed we have the following corresponding tree:



After transformation using the l.f.m.s. of the <do-group> we obtain the following tree:



6. Procedure

6.1 Main Procedure Heading

a) The main procedure heading consists of the statement

procedure main

followed by a list of attribute statements. At this level, as all attribute statements of the procedure were considered, we can put all attributes of a given subject in one declaration tree labeled def (for default tree) (H_{2b}, H_{4b}). Then we can verify that a type is given to each subject (H_{2a}, H_{4a}).

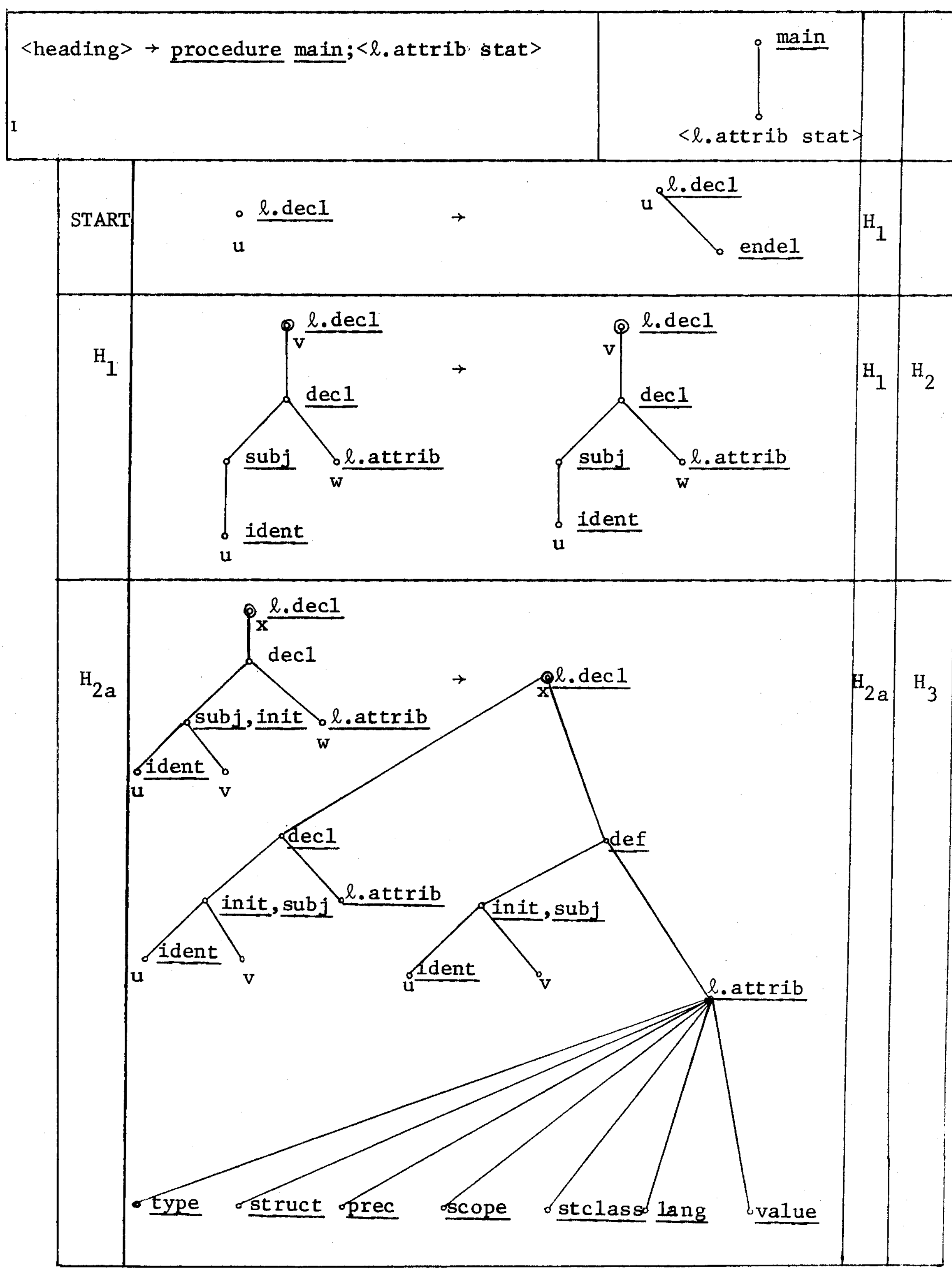
But an important problem arises: we should not lose the order in which the declarations, more precisely the initializations, occur. This is done by looking first for the initializations from left to right (H_1, H_2), and leaving the declarations without initializations for another scan (H_3, H_4).

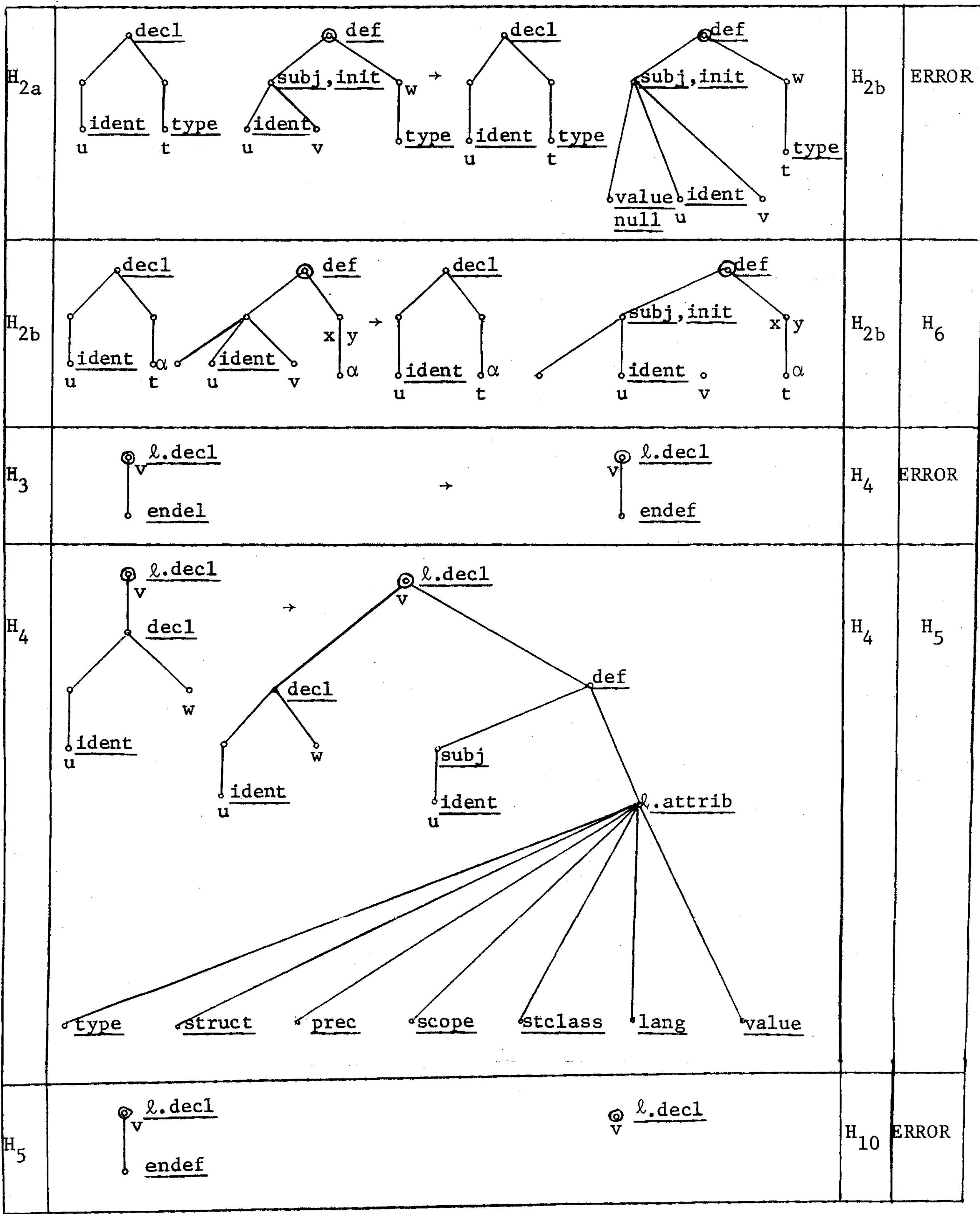
After considering a declaration, its deletion is performed (H_{6a}, H_{6b}). Also a procedure specification that has no type (i.e. is not a function but a subroutine) has all its attributes except the language attribute deleted as they are not useful (H_7).

The assignment of default attributes follows (H_{8a}, H_{8b}). A function def is used to give for each attribute α its default for variables. But for a function the default for the precision attribute is long while it is short for ordinary variables.

All this is repeated for all declarations (H_9). Then when there is no more declaration trees, some incompatibilities of attributes should be detected (H_{11a}, H_{11b}, H_{11c}, H_{12}).

b)

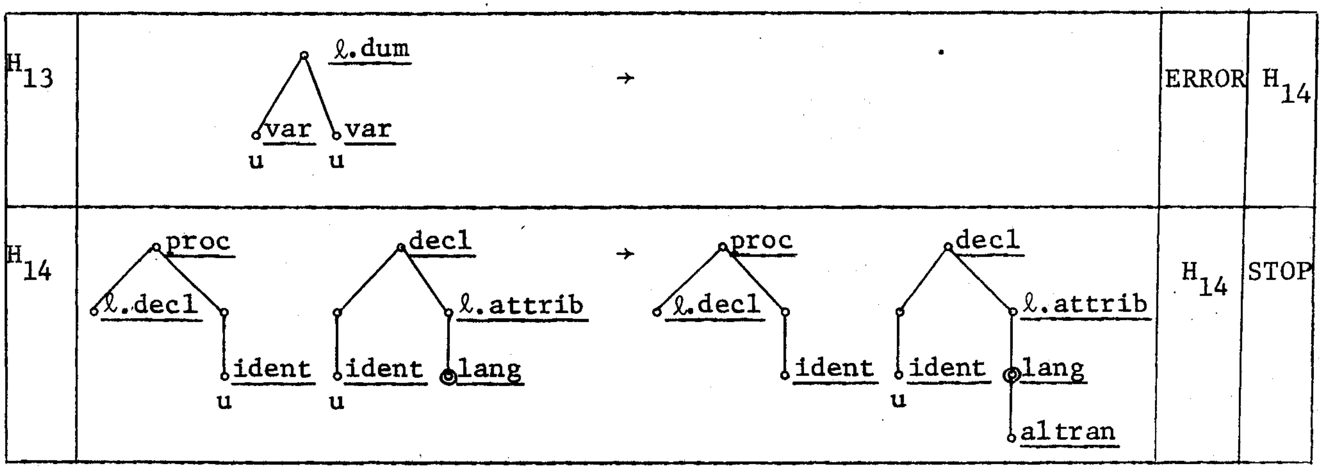




H2a
decl
ident
type
u
t
def
subj,init
ident
w
type
v
value
null
ERROR
H2b
H6
α
x
y
H3
ℓ.decl
endel
endef
H4
ERROR
decl
H5
subj
ℓ.attrib
struct
prec
scope
stclass
lang
H10

H_{4a}	decl; ident u; type t ◎ def; subj; w; ident u; type → decl; ident u; type t ◎ def; subj; w; ident u; type t	H_{4b}	ERROR
H_{4b}	decl; ident u; α t ◎ def; subj; x | y; ident u; α → decl; ident u; α t ◎ def; subj; x | y; ident u; α t	H_{4b}	H_6
H_{6a}	◎ decl; x | y; w; ident u def; ident u → ◎ decl def; ident u	H_{6a}	H_{6b}
H_{6b}	◎ ℓ.decl; decl → ◎ ℓ.decl	H_{6b}	H_7
H_7	◎ ℓ.attrib; v; w; type; lang; altran → ◎ ℓ.attrib; lang; altran	H_7	H_{8a}
H_{8a}	def; u; x; z; y; prec; long; altran → ◎ def; u; x; z; y; prec; lang; long; altran	H_{8a}	H_{8b}

H_{8b}	def; u; x | y; α_1 → def; u; x | y; α_1; def(α_1) $\alpha_1 \in$ {prec,struct,scope,st class}	H_{8b}	H_9
H_9	ℓ.decl; endcl → ℓ.decl; endcl	H_1	H_4
H_{10}	def → decl	H_{10}	H_{11a}
H_{11a}	ℓ.attrib; type; prec; log; long →	ERROR	H_{11b}
H_{11b}	ℓ.attrib; label; long →	ERROR	H_{11c}
H_{11c}	ℓ.attrib; scope; st class; external; autom →	ERROR	H_{12}
H_{12}	decl; ℓ.attrib; label; external →	ERROR	H_{13}



def: {struct,prec,scope,st class} → {scalar,short,internal,autom}

def(struct) = scalar

def(prec) = short

def(scope) = internal

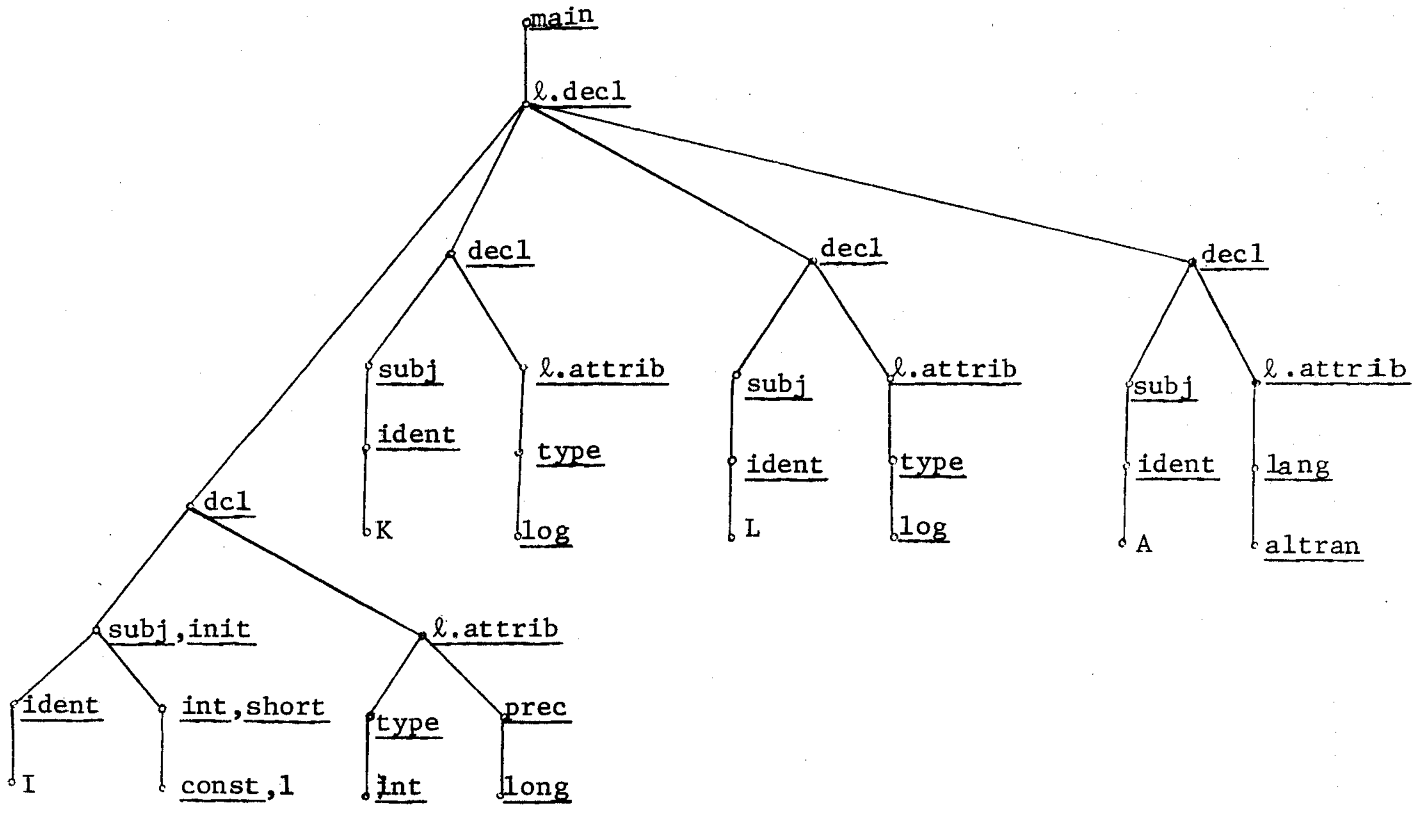
def(st class) = autom

c) Consider the following heading of a main procedure.

procedure main;

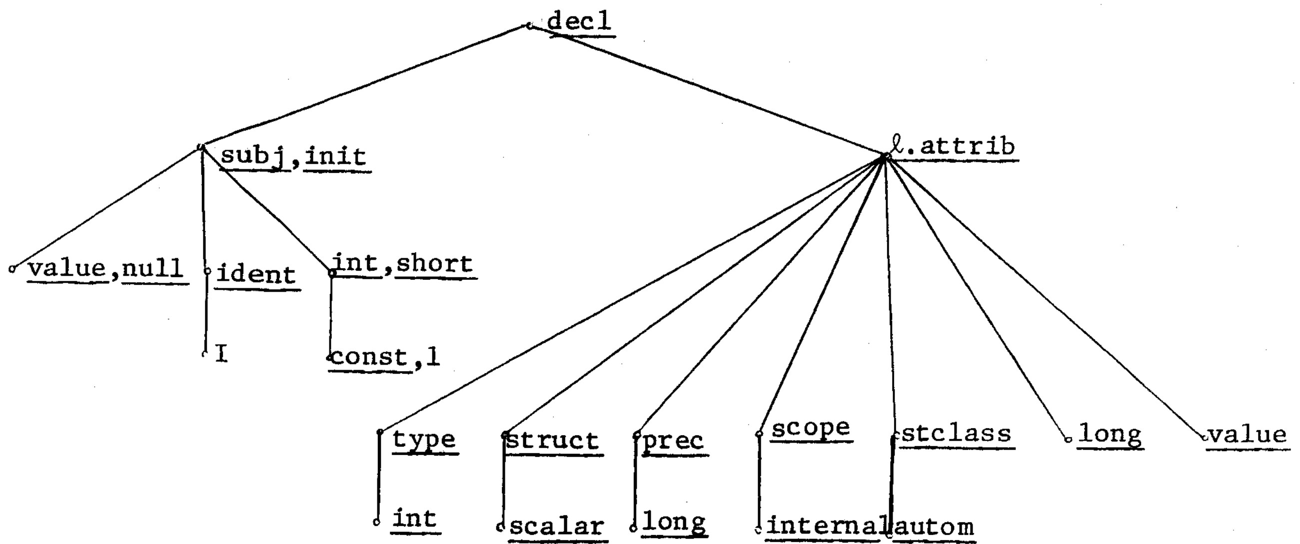
long integer I = 1; logical K,L; altran A

Before using the l.f.m.s. to transform it the original tree representing this heading would be as follows:



After performing the transformations using the l.f.m.s. the declaration trees of I, K and L change their structure while that of A has only a new branch for the value added.

The declaration tree of I becomes:



6.2 Procedure Heading

a) The heading of a non-main procedure gives a name to the procedure and eventually a list of dummy variables. A list of attribute statements follows in general.

The l.f.m.s. for the procedure heading is the same as that of the main procedure. So, for schemata H_1 to H_{12} the informal description of the main procedure heading is the same as here.

The two supplementary productions (H_{13}, H_{14}) check that no dummy variable appears more than once in the list of dummies (H_{13}) and give the language attribute altran to a declaredvariable which is the procedure name (H_{14}). This transformation is necessary when a function procedure calls itself and has to be given a type. The language attribute that remained empty is filled with the attribute altran showing that the variable is a procedure name.

b)

<heading> → <u>procedure</u><ident>(<ℓ.dum>); <ℓ.attrib stat> 1	<ℓ.attrib stat> <ident> <ℓ.dum>

START H_1 ⋮ H_{14}	The same as for the heading of the main procedure	H_1 H_1 ⋮ H_{14}	H_2 ⋮ STOP

<table>
<tr><td colspan="3"><heading> → <u>proc</u> <ident>;<ℓ.attrib stat>

2</td><td colspan="2"><ℓ.attrib stat>
<ident>
<u>ℓ.dum</u>
<u>blank</u></td></tr>
<tr><td>START
H_1
⋮
H_{14}</td><td colspan="2">As for the heading of the main procedure</td><td>H_1
H_1
⋮
H_{14}</td><td>
H_2
⋮
STOP</td></tr>
</table>

6.3 Procedure

6.3.1 Procedure Structure

a) When constructing the tree for a procedure we have to add a branch with a node labeled <u>staint</u> for the <u>static</u> <u>internal</u> variables. At this node we shall attach later on the declaration trees of variables having the attributes <u>static</u> <u>internal</u>.

The l.f.m.s. checks first for the duplication of any indeterminate or label declared implicitly (START, K_1).

Then we must make sure that a variable that is used as <u>short</u> and/or <u>int</u> (e.g. control variable) is declared to be <u>short</u> and/or <u>int</u> (K_{2a}, K_{2b}).

Now we link each used variable to the value node of its declaration (K_{3a}, K_{3b}, K_{3c}). For the particular case of an indeterminate its value is its name. Consequently, the name of a variable indeterminate is not deleted while the name of the other types of variables is deleted from the usage tree. Also a pointer to a layout in which the indeterminate was declared is constructed.

The case of static internal variables is resolved by making a copy of the declarations of these variables and attaching them to a special node of the procedure tree that we labeled staint. A link is kept with the original declaration tree for initialization purposes, while all variables will point to the copy of the declaration tree instead of the original one (K_{3d}-K_{3e}).

No operations on arrays as a whole are allowed (K_{3f}).

The algebraic variables should also have a pointer to their layout constructed (K_{3g}).

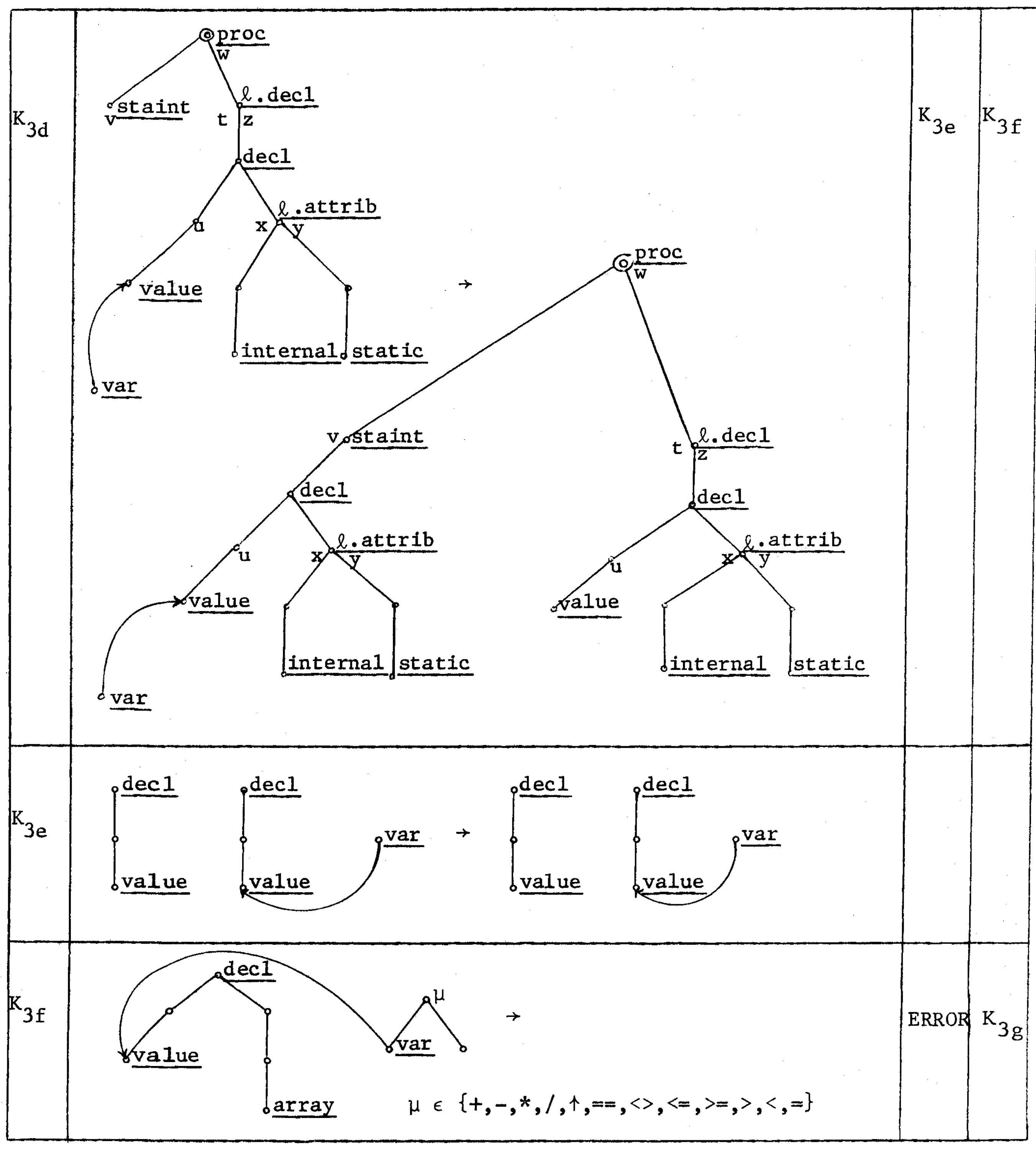
A dummy label should not be used within a procedure for other purposes than as a return point, and any label variable appearing in a return-statement should be a dummy label (K_{4a}, K_{4b}, K_{4c}).

If there is an invocation of a procedure in the procedure itself then a pointer is constructed from the invocation to the procedure tree root labeled proc (K_{5a}). Also if there is a specification of the procedure then a pointer to this specification is constructed. If neither of these two situations occur there is an error (K_{5a}, K_{5b}, K_{5c}).

b)

<proc> → <heading>;<body>;<end stat> 1		proc <heading> staint <body>		
START	u dcl u dcl	→	ERROR	K_1
K_1	u dcl decl subj u	→	ERROR	K_{2a}

K_{2a}	decl; ident (u), prec — long var — used,short — ident →	ERROR	K_{2b}
K_{2b}	decl; ident (u), type — τ_1 var — used,int — ident (u) →	ERROR	K_{3a}
K_{3a}	decl; value, ident (u), type — τ, prec — π var — used — ident (u) → decl; value, ident (u), type — τ, prec — π; var, τ, π	K_{3a}	K_{3b}
K_{3b}	label,dcl (u) used,label — ident → label,dcl; label	K_{3b}	K_{3c}
K_{3c}	alg,layout — indet,dcl used — ident (u) → alg,layout — indet,dcl (u); indet,ident	K_{3c}	K_{3d}



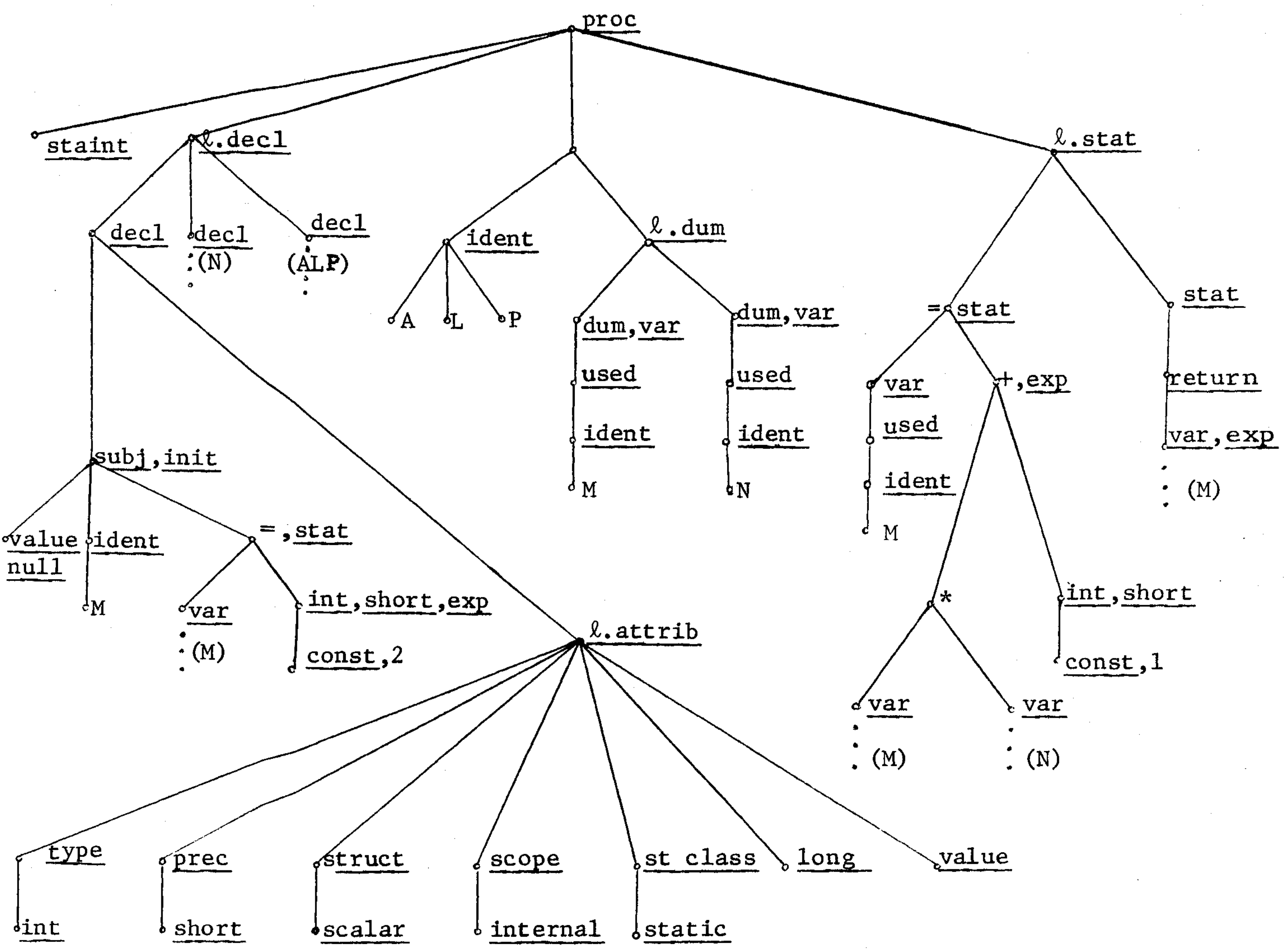
K_{3d}
proc
w
v staint
t
ℓ.decl
z
decl
u
x
ℓ.attrib
y
value
internal
static
var
→
proc
w
v staint
t
ℓ.decl
z
decl
u
x
ℓ.attrib
y
value
internal
static
var
decl
u
x
ℓ.attrib
y
value
internal
static
K_{3e}
K_{3f}
K_{3e}
decl
value
decl
value
var
→
decl
value
decl
value
var
K_{3f}
decl
value
array
var
μ
→
μ ∈ {+,-,*,/,↑,==,<>,<=,>=,>,<,=}
ERROR
K_{3g}

K_{3g}	decl, value, type, alg, alg,var → decl, value, type, alg, alg,var	K_{3g}	K_{4a}
K_{4a}	value, dum,var, dumlab,var → value, dum,var	K_{4a}	K_{4b}
K_{4b}	return,dumlab, value →	ERROR	K_{4c}
K_{4c}	value, dum,var, var →	ERROR	K_{5a}
K_{5a}	invoc, ident, u, v; proc, ident, u, w → proc, invoc, v, ident, u, w	K_{5a}	K_{5b}
K_{5b}	invoc, ident, u, v; decl, ident, u, ℓ.attrib, lang, altran → decl, invoc, v, ident, u, ℓ.attrib, lang, altran	K_{5b}	K_{5c}
K_{5c}	invoc, ident, u, v →	ERROR	K_6

c) Consider the following procedure function:

```
procedure  ALP (M,N);
      int  M,N,ALP; static  M = 2;
        M = M * N+1; return (M);
end
```

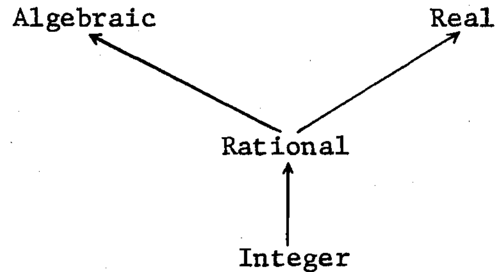
This procedure will have the following tree as representation before applying to it the transformations performed by the l.f.m.s.



Note that for ALP the precision attribute will be long and not short as it is for M or N, because ALP is the name of a function procedure.

6.3.2 Compatibility of types in an arithmetic expression

a) At this level as the types of the variables are known, we can check the compatibility of types in an arithmetic expression. A search is performed in any tree representing an expression. During this search the type of each variable or constant is examined and a type for each subexpression is deduced. This type is in fact the highest of the types that appear in this expression according to the following hierarchy of types.



Note that algebraics and reals are not comparable in this hierarchy as they are not compatible. Thus we must check that algebraics and reals are not mixed in an expression.

Starting at the production labeled K_6 an expression is "picked up" and two control words VER (for verify) and ANY (for any type) will direct our search in the tree. This search will be performed from left to right and the label VER will push the search downwards while REV (for reverse) will push it upwards.

The control label ANY will be replaced by one of the control labels INT, RAT, ALG and REAL depending on the first variable encountered. If at

some point the label ALG appears, a real variable should never be encountered. Also if at some stage the label REAL appears, an algebraic variable should never be encountered.

If several algebraics (including indeterminates) appear in an expression, they should have the same layout. But the checking for this constraint is deferred to the semantics phase mainly because of substitution whose layout will not be known before the interpretation phase.

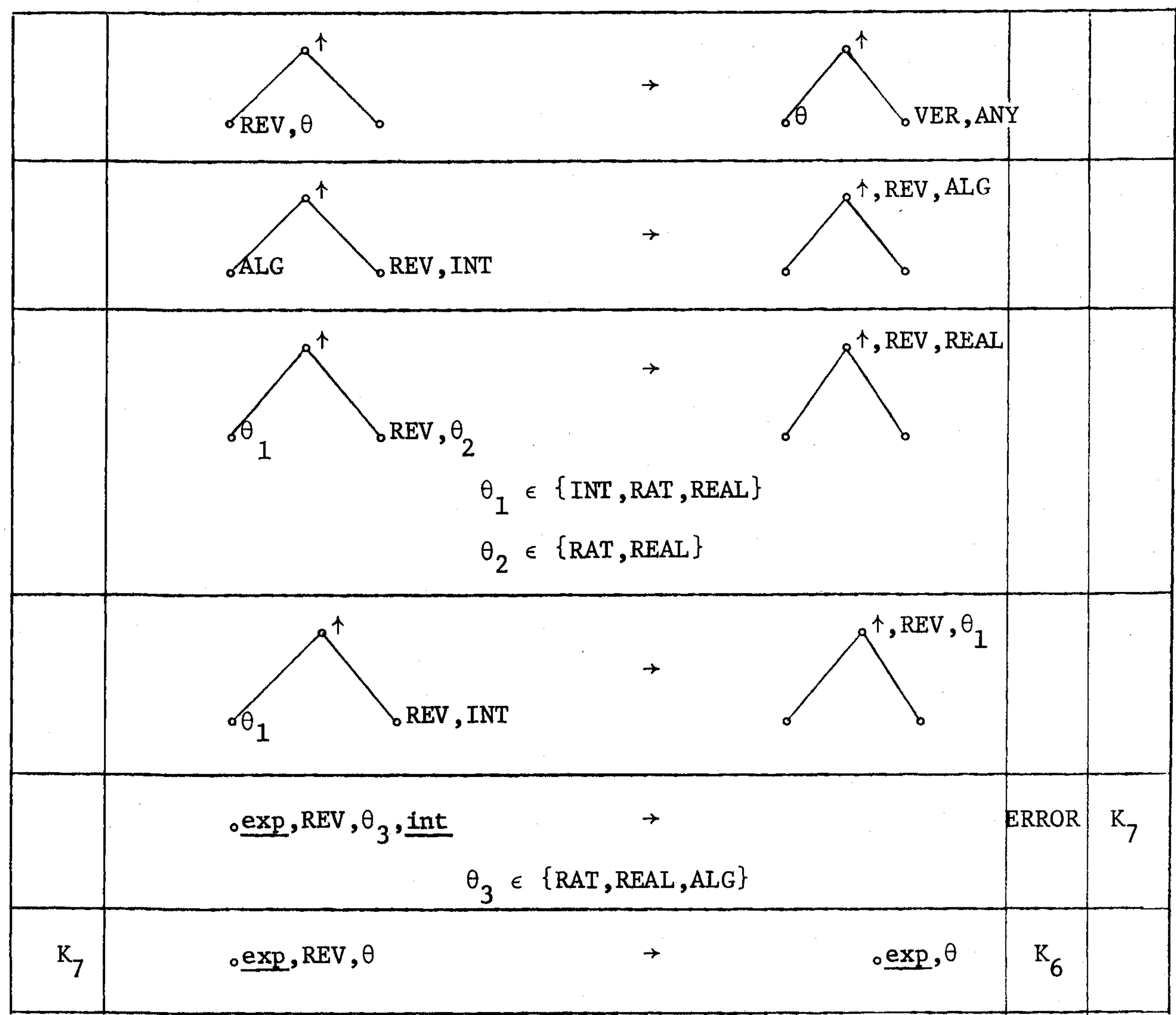
The restrictions on exponentiation are also verified using five supplementary productions. In fact, there is more restrictions on exponentiation, but they can only be verified at the semantics level. Here we only make sure that if the base is algebraic then the exponent is integer.

b)

K_6	∘<u>exp</u> → ∘<u>exp</u>,VER,ANY		K_{7a}
	⊖,VER,ANY → ⊖ VER,ANY		
	⊖ ,VER,θ → ⊖ VER,θ θ ∈ {INT,RAT,REAL,ALG}		
	ν,VER,ANY → ν VER,ANY ν ∈ {+,-,*,/}		

	elem, VER, ANY (node with two sons) → elem (node with two sons; left son: VER, ANY)		
	ν, VER, θ (node with two sons) → ν (node with two sons; left son: VER, θ)		
	elem, VER, θ (node with two sons) → (node with two sons; left son: VER, θ)		
	∘alg, VER, η → ∘alg, REV, ALG $\eta \in$ {ANY, INT, RAT}		
	∘indet, VER, η → ∘indet, REV, ALG		
	∘subst, VER, η → ∘subst, REV, ALG		
	∘real, VER, η → ∘real, REV, REAL		
	∘rat, VER, η → ∘rat, REV, RAT		
	∘int, VER, η_1 → ∘int, REV, INT $\eta_1 \in$ {ANY, INT}		
	∘β, VER, ALG → ∘β, REV, ALG $\beta \in$ {int, rat, indet}		

	$\circ\beta_1$,VER,REAL	→	$\circ\beta_1$,REV,REAL		
	$\beta_1 \in \{$<u>int</u>,<u>rat</u>$\}$				
	$\circ$<u>int</u>,VER,RAT	→	$\circ$<u>int</u>,REV,RAT		
	ν (left: REV,θ; right: —)	→	(left: —; right: VER,θ)		
	ν (left: —; right: REV,θ)	→	ν,REV,θ (left: —; right: —)		
	Θ (child: REV,θ)	→	Θ,REV,θ (child: —)		
	<u>elem</u> (left: REV,θ; right: —)	→	<u>elem</u>,REV,θ (left: —; right: —)		
	$\uparrow$,VER,θ (left: —; right: —)	→	$\uparrow$ (left: VER,θ; right: —)		
	$\uparrow$,VER,ANY (left: —; right: —)	→	$\uparrow$ (left: VER,ANY; right: —)		
	$\uparrow$ (left: REV,θ; right: —)	→	$\uparrow$ (left: θ; right: VER,ANY)		



6.3.3 Compatibility of types in the assignment statement

a) For the arithmetic assignment statement the error of assigning a real expression to an integer, a rational or an algebraic variable is detected.

b)

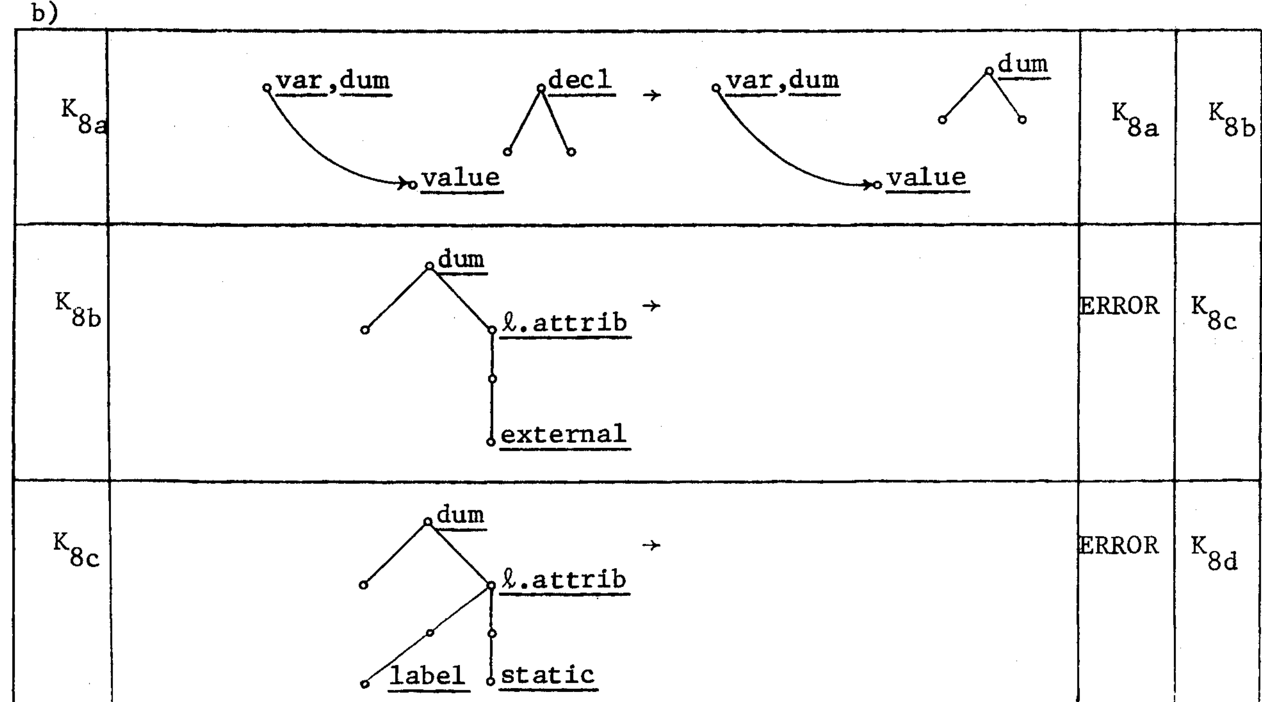
K_{7a}	= alg REAL	→	ERROR	K_{7b}
K_{7b}	= rat REAL	→	ERROR	K_{7c}
K_{7c}	= int REAL	→	ERROR	K_{8a}

6.3.4 Some restrictions related to dummy variables

a) These restrictions are:

- A dummy cannot be external (K_{8b})
- A dummy label cannot be static (K_{8c})
- A dummy label is given automatically the value attribute (K_{8d})
- No variable other than dummy can possess the value attribute (K_{8e})

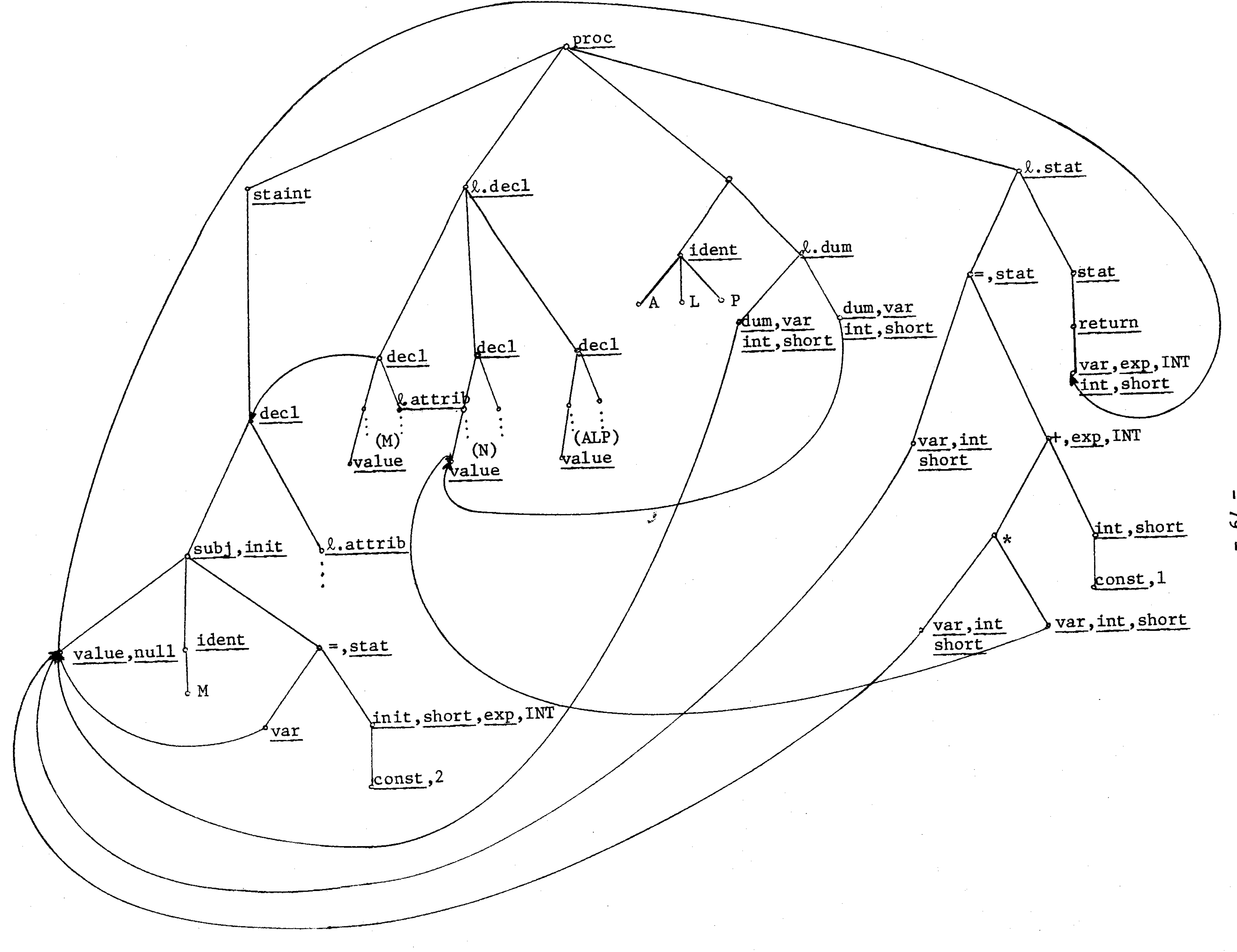
b)



K_{8d}	dum; ℓ.attrib; type; val; label → dum; ℓ.attrib; val; label; value	K_{8d}	K_{8e}
K_{8e}	decl; ℓ.attrib; val; value →	ERROR	K_{8f}
K_{8f}	dum; ℓ.attrib → decl; ℓ.attrib	STOP	STOP

c) Consider again the procedure given in 6.3.1c but after performing the transformations (l.f.m.s.). The following changes would be noticed. First, there will be a copy of the declaration tree of M attached to the node labeled staint. Second, all variables including dummies have a link to the value node of their declaration with their name is deleted from the usage subtree. Third, on the top node of each expression (labeled exp) we have the type of the expression (INT, REAL, ...).

All these changes appear on the following tree.



proc
staint
ℓ.decl
ident
A
L
P
ℓ.dum
dum,var
int,short
dum,var
int,short
ℓ.stat
=,stat
stat
return
var,exp,INT
int,short
decl
decl
decl
ℓ.attrib
(M)
value
(N)
value
(ALP)
value
decl
var,int
short
+,exp,INT
subj,init
ℓ.attrib
int,short
const,1
*
value,null
ident
M
=,stat
var,int
short
var,int,short
var
init,short,exp,INT
const,2

7. Program

a) A program consists in general of several procedures one of them being a main procedure.

The global variables (external static) as well as the algebraic options are considered in the l.f.m.s.

The same technique as for internal static variables is used here. A copy is made of the declaration and attached to the glob-var node. A link between the original declaration and the copy is kept for initialization purposes. The variables pointing to the original declaration will point now to the copy (M_1,M_2).

All declarations of the same global variable, which should have the same attributes, will be linked to the global tree thus constructed (M_3,M_4).

For all specifications of subprocedures in a procedure heading, construct the link between the specification and the procedure itself, while deleting the name from the declaration (M_{5a}).

Naturally, each specification of a subprocedure name should be related to a procedure tree otherwise an error is detected (M_{5b}).

Function invocations will be linked to the function definition directly while keeping the link to the specification. The subroutines will also be linked to their definition but we do not need keeping any link to the specification as it will be deleted next (M_{5d},M_{6a}).

Finally the names of the procedures as well as the names of the declared variables (appearing in the declarations) will be deleted (M_{6b},M_7,M_8,M_9).

b)

<prog> → <proc> 1	prog <proc>
<prog> → <prog>;<proc> 2	<prog> <proc>

START	prog main,proc	→	prog main,proc	M	ERROR
M	prog main main	→		ERROR	M
M_0	prog u	→	prog u glob var 101 102 103 1 1 1	M_1	
M_1	prog x y globvar v proc w ℓ.decl x_1 x_2 decl subj u value var ℓ.attrib t z scope external		prog v globvar decl subj u value var ℓ.attrib t z scope external proc w ℓ.decl x_1 x_2 decl subj u value ℓ.attrib t z scope external	M_2	M_{4a}

M_2	decl value decl value var → decl value decl value var	M_2	M_4
M_3	glob var decl ident u ℓ.attrib w decl ident u ℓ.attrib v → glob var decl ident ℓ.attrib decl ident u ℓ.attrib v	M_4	M_1
M_4	glob var decl ident u w decl ident u w → glob var decl ident u w decl ident w	M_2	ERROR
M_{4a}	decl value elem var,τ,π → decl value elem,τ,π var,τ,π	M_{4a}	M_{5a}
M_{5a}	proc ident u decl x y t z ident u ℓ.attrib altran → proc ident u decl xy t z altran	M_{5a}	M_{5b}
M_{5b}	decl ident u ℓ.attrib altran →	ERROR	M_{5c}

M_{5c}	proc, decl, invoc,fnct → proc, decl, invoc,fnct	M_{5c}	M_{5d}
M_{5d}	proc, decl, invoc,stat → proc, decl, invoc,stat	M_{5d}	M_{6a}
M_{6a}	ℓ.decl, x y, decl, v, ℓ.attrib, altran → ℓ.decl, xy	M_{6a}	M_{6b}
M_{6b}	proc, v, ident, u → proc, v	M_{6b}	M_7
M_7	del, u →	M_7	M_8
M_8	decl, subj,init, ℓ.attrib, w, null, ident, u, v → decl, subj,init, ℓ.attrib, w, null, v	M_8	M_9
M_9	decl, subj, ℓ.attrib, w, null, ident, u → decl, subj, ℓ.attrib, w, null	M_9	STOP

8. The Data Language

a) For the input as well as the output a data language exists. An expression in this data language consists of combination of constants using the elementary arithmetic operations. A statement in this language is a data expression or a constant including the null constant.

Some notation using backlabeling is defined and an l.f.m.s. is used to transform the corresponding tree to one reflecting its meaning but using only the elementary operations on expressions.

b)

1 <data prim> → <int const>	∘ <int const>
2 <data prim> → (<data exp>)	∘ <data exp>
3 <data fact> → <data prim>	∘ <data prim>
4 <data fact> → <data prim>**<int const>	↑ <data prim> <int const>
5 <data fact>→<data prim>**(<s int const>)	↑ <data prim> <s int const>
6 <data term> → <data fact>	∘ <data fact>
7 <data term>→<data fact>*<data fact>	* <data fact> <data fact>

<data term> → <data fact>/<data fact> 8	/ <data fact> <data fact>
<data exp> → <data term> 9	<data term>
<data exp> → +<data term> 10	<data exp>
<data exp> → −<data term> 11	⊖ <data term>
<data exp>→<data exp>+<data term> 12	+ <data exp> <data term>
<data exp> → <data exp>−<data term> 13	− <data exp> <data term>
<data stat> → <data exp> 14	<data exp>
<data stat> → <real const> 15	<real const>
<data stat> → +<real const> 16	<real const>
<data stat> → −<real const> 17	⊖ <real const>
<data stat> → <log const> 18	<log const>

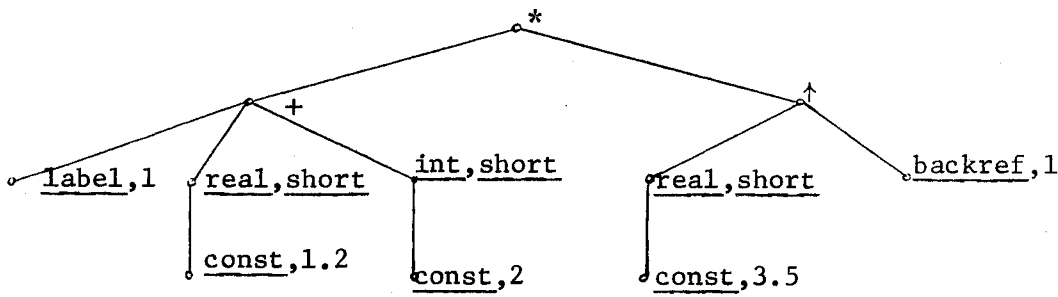
No.	Production	Graph
19	<data stat> → null	∘null
20	<input> → ε	∘input
21	<input> → <data stat>	∘input │ ∘data stat <data stat>
22	<input> → <input>;<data stat>	∘ <input> \\ ∘data stat <data stat>
23	<output> → ε	∘output
24	<data prim> → <back ref>	∘ <back ref>
25	<back ref> → (=<ilabel>)	∘back ref <ilabel>
26	<ilabel> → <int const>	∘ <int const>
	START: ◎int │ ∘const,χ → ◎ χ	START STOP
27	<data exp> → <ilabel>:<data exp>	∘<data exp> / ∘ilabel <ilabel>
28	<data stat> → <data exp>	∘ <data exp>

START	◎back ref,χ ∘ilabel,χ (u)	→	◎ (u) ∘ilabel,χ (u)	START	D_1
D_1	◎ — ∘ilabel,χ	→	◎	D_1	D_2
D_2	∘back ref,χ	→		ERROR	STOP

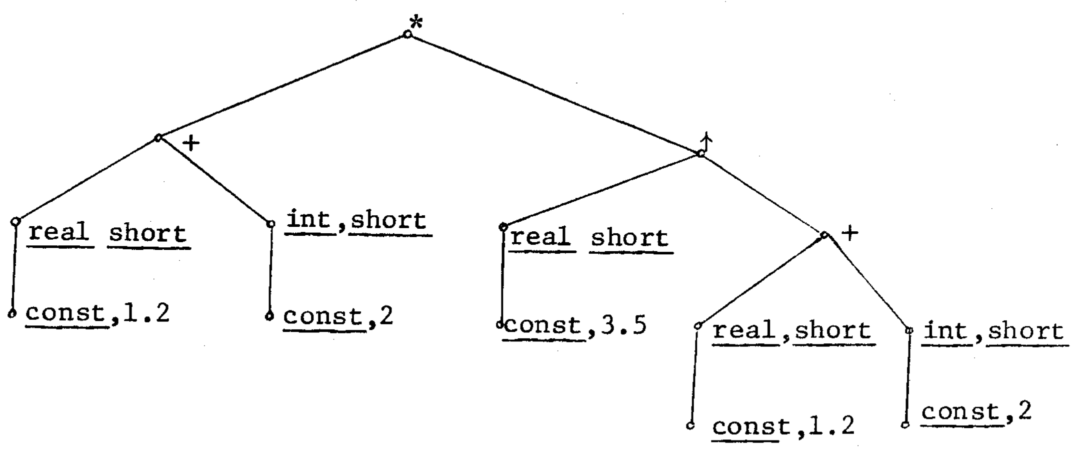
c) Consider the following data expression

(1:(1.2+2))*3.5**(=1)

Before applying the l.f.m.s. of the <data stat> to it we have:



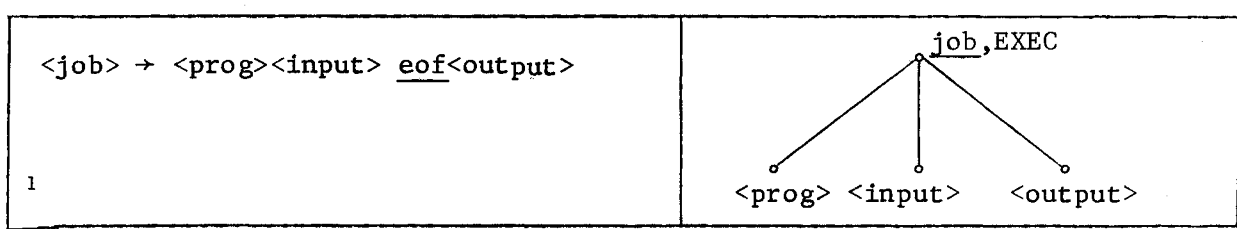
After transformation by the l.f.m.s. we would have:



9. <u>Job</u>

a) A job consists of the program followed by the input and the output, if any. The input is separated from the output by the word <u>eof</u> (end of file). The control word EXEC is attached to start the interpretation of the job, which leads us to the semantics description.

b)



II. The Semantics Description

1. Initialization

a) The execution of a job starts at the main procedure and ends when the end of this main procedure is reached (1-4). But the execution of any procedure requires first the initialization of some of its variables. That is why a control word INIT is used to go through the list of declarations and initialize those scalar variables that need initialization. Array variables cannot be initialized, but we must construct the array and attach it to the value node using a macro constr (19) defined below.

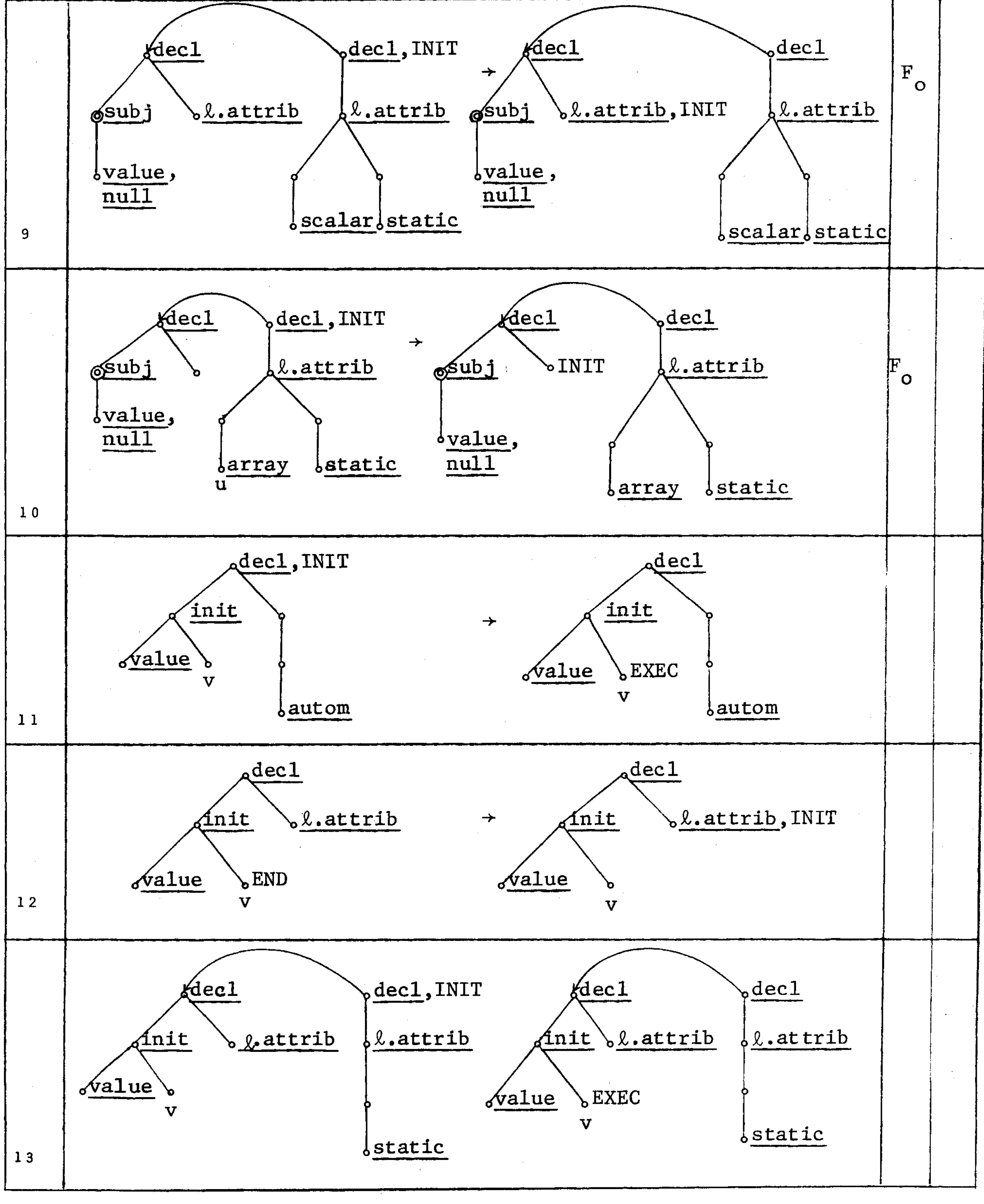
The initialization of automatic variables should be performed each time we enter the procedure. But the initialization of static variables is done only at the first call of the procedure and if its value is null (for it can be external and have a value from another procedure). This is performed in productions (7-13).

On the other hand, for the arrays and algebraics the expressions in the descriptor block (D.B.) or the layout should be computed once for all for that invocation (14-18).

b)

START 1	job,EXEC prog	→	job prog,EXEC		
2	prog,EXEC proc,main	→	prog proc,main,EXEC		

3	prog proc, main, END	→	prog, END proc, main		
4	job prog, END	→	job prog	STOP	
5	proc, EXEC ℓ.decl	→	proc ℓ.decl, INIT		
6	ℓ.decl, INIT u decl	→	ℓ.decl u decl, INIT		
7	decl, INIT subj — ℓ.attrib value null — scalar, autom	→	decl subj — ℓ.attrib, INIT value null — scalar, autom	F_o	
8	decl, INIT subj — ℓ.attrib value, null — array u, autom	→	decl subj — ℓ.attrib, INIT value null — array u, autom	F_o	



9
decl
decl, INIT
subj
ℓ.attrib
ℓ.attrib
value, null
scalar
static
→
decl
decl
subj
ℓ.attrib, INIT
ℓ.attrib
value, null
scalar
static
F o
10
decl
decl, INIT
subj
ℓ.attrib
value, null
array
u
static
→
decl
decl
subj
INIT
ℓ.attrib
value, null
array
static
F o
11
decl, INIT
init
value
v
autom
→
decl
init
value
EXEC
v
autom
12
decl
init
ℓ.attrib
value
END
v
→
decl
init
ℓ.attrib, INIT
value
v
13
decl
decl, INIT
init
ℓ.attrib
ℓ.attrib
value
v
static
decl
decl
init
ℓ.attrib
ℓ.attrib
value
EXEC
v
static

F_o 14	ℓ.attrib, INIT / struct / array / ∘ / exp	→	ℓ.attrib, INIT / struct / array / ∘ / exp, EXEC	F_1
F_1 15	ℓ.attrib, INIT / type / alg / ∘ / exp	→	ℓ.attrib / type / alg / ∘ / EXEC	F_2
16	alg / ◎ / u v / VAL, p	→ $p \in Z^+$	alg, ENDIN / ◎ / u p	
17	array / ∘ / u ◎ / VAL, n	→ $n \in Z$	array, ENDIN / ∘ / ◎ n	
18	ℓ.attrib / ∘ / ENDIN	→	ℓ.attrib, INIT / ∘ / ∘	

F_2 19	decl; ℓ.attrib, INIT; value; array; u	→	decl; ℓ.attrib, INIT; value; constr(u); array	F_2	F_3
20	ℓ.decl; u; v; decl, END; decl	→	ℓ.decl; u; v; decl; decl, INIT		
21	u; ℓ.decl; decl, END	→	ℓ.decl, END; u; decl		
22	globvar; decl; decl; init; ℓ.attrib; value; u; v	→	globvar; decl; decl; ℓ.attrib; value; u		
23	staint; decl; decl; init; ℓ.attrib; value; u; v	→	staint; decl; decl; ℓ.attrib; value; u		
F_3 24	decl; ℓ.attrib, INIT	→	decl; ℓ.attrib, END		

The macro constr is defined by the following productions:

START	top u	→	top u CONS HERE	M_1	
M_1	CONS n n HERE u $n \in Z$	→	CONS n n u n	M_3	M_2
M_2	CONS m n HERE u $m \in Z$	→	CONS add(m,1) n HERE u m	M_1	
M_3	u CONS n n k ℓ HERE v $k,\ell \in Z$	→	u CONS k ℓ v HERE	M_1	M_4
M_4	top u CONS n n	→	top u	M_5	
M_5	u v HERE w	→	u v w HERE w	M_5	M_6
M_6	u m,HERE	→	HERE u m	M_5	M_7
M_7	top HERE u	→	u	STOP	

2. List of statements

a) A list of statements is executed from left to right. When there is no more statements to be executed the procedure ends execution.

b)

1	proc; ℓ.decl,END; ℓ.stat	→	proc; ℓ.decl; ℓ.stat,EXEC		
2	ℓ.stat,EXEC; u	→	ℓ.stat; u; EXEC		
3	ℓ.stat; u: END; v: .	→	ℓ.stat; u; v: EXEC		
4	ℓ.stat; u: END	→	ℓ.stat,END; u		
5	proc,main; ℓ.stat,END	→	proc,main,END; ℓ.stat		

3. The continue, goto, if, doend and end statements

a) The continue, doend, and end-statements have no effect at all (1-4). The goto and the if-statements have an obvious meaning (5-9).

b)

1	ℓ.stat; continue, EXEC	→	ℓ.stat; continue,END		
2	ℓ.stat; exit,EXEC	→	ℓ.stat,END; exit		

3	ℓ.stat doend,EXEC	→	ℓ.stat doend,END		
4	ℓ.stat end,EXEC	→	ℓ.stat end,END		
5	goto,EXEC value stat	→	goto value stat EXEC		
6	goto,EXEC stat	→	goto stat,EXEC		
7	if,EXEC	→	if EXEC		
8	if LOG VAL,true	→	if EXEC		
9	if LOG VAL,false	→	if EXEC		

4. The return-statement (refer to 12.1 when necessary)

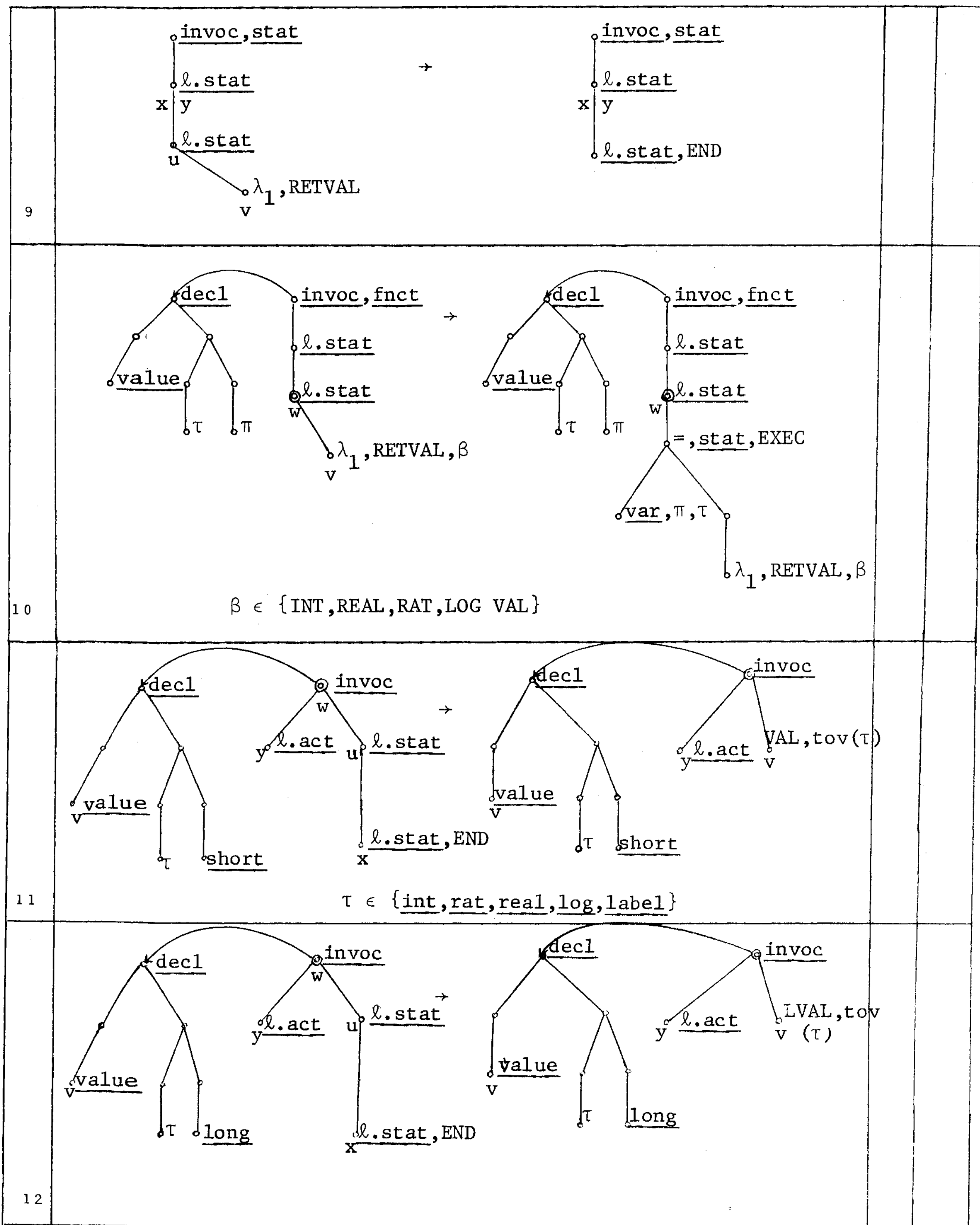
a) The return-statement in its simplest form, return, leads the execution out of the body of the procedure to perform the outbound transmission (1-4).

But when it has an expression, arithmetic or logical as argument, the return-statement becomes more complicated to describe. First, we have to execute the expression (5). Whenever the value has been found we have to move it upwards the tree till it reaches the node labeled invoc (6-8). If the invocation is a statement (subroutine invocation), this value is needless; we just delete it (9). Otherwise (it is a function invocation), an assignment statement is constructed to assign the returned value to the variable created in this purpose (10). After the execution of this assignment statement the control word EXEC will appear at the outbound transmission subtree. But in the case of a function we have to transmit the value of this function to be used in the expression and delete the copy of the procedure. If the function is algebraic we have to transmit also the layout with the value (11-14).

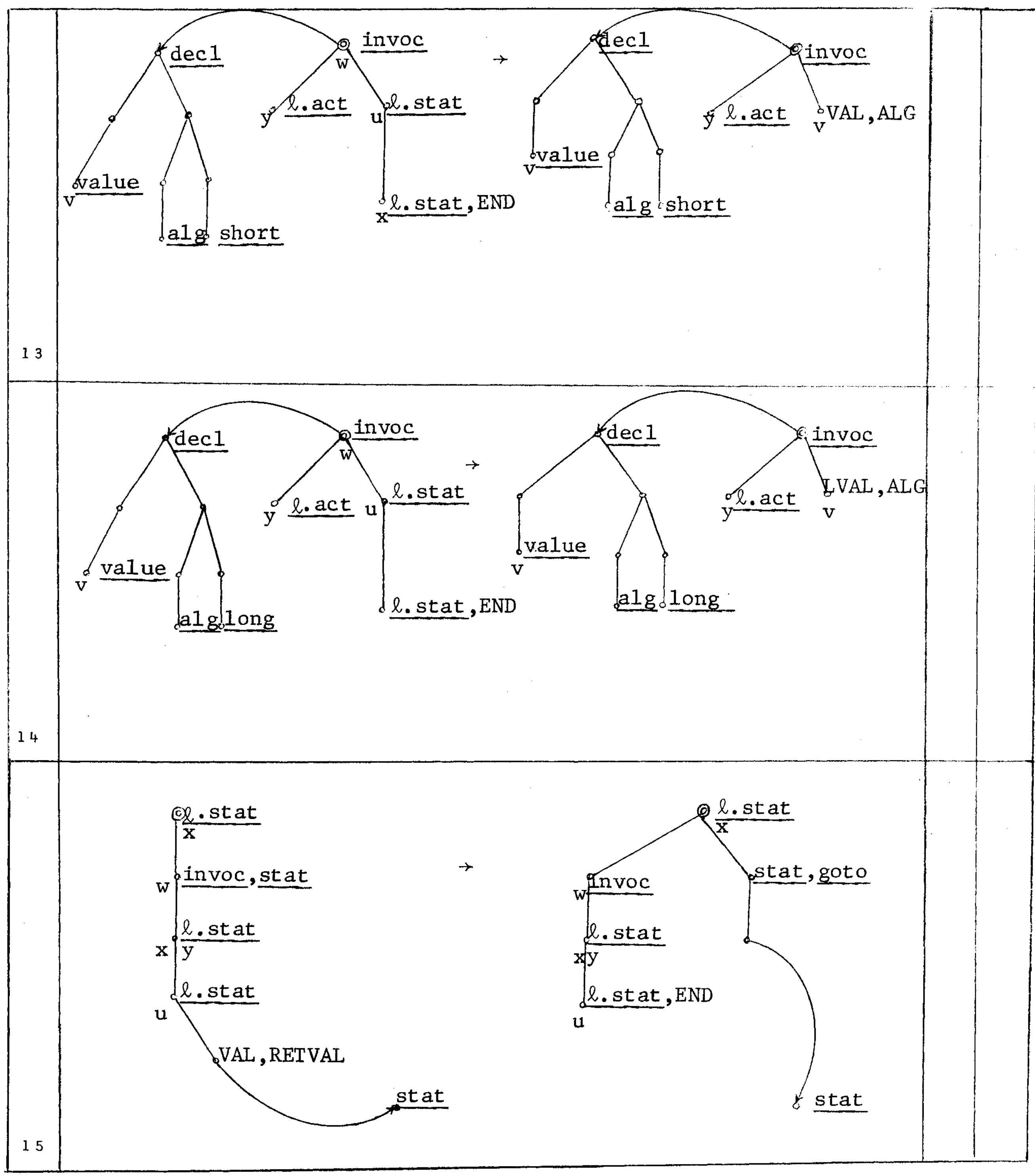
The third type of return-statement is that with label argument. This third type can be used only in a subroutine procedure and it means that instead of returning to the invocation point in the invoker a jump is executed to some statement given by the label argument (15). Note that the outbound transmission should be executed before this return is made.

b)

1	stat return,EXEC	→	stat,RET return	
2	ℓ.stat stat,RET	→	ℓ.stat,RET stat	
3	stat ℓ.stat,RET	→	stat,RET ℓ.stat	
4	ℓ.stat u v ℓ.stat,RET ℓ.stat	→	ℓ.stat u v ℓ.stat ℓ.stat,EXEC	
5	return,EXEC ξ	→	return ξ,EXEC	
		$\xi \in$ {exp,log exp}		
6	stat return u λ_1,RETVAL	→	stat return λ_1,RETVAL u	
7	return ξ u λ_1 v	→	return ξ u λ_1,RETVAL v	
		$\lambda_1 \in$ {VAL,LVAL,LOGVAL}		
8	ℓ.stat stat u λ_1,RETVAL v	→	ℓ.stat stat λ_1,RETVAL u v	



invoc,stat
ℓ.stat
x y
ℓ.stat
u
λ1,RETVAL
v
9
→
invoc,stat
ℓ.stat
x y
ℓ.stat,END
decl
invoc,fnct
ℓ.stat
value
ℓ.stat
w
τ π
λ1,RETVAL,β
v
→
decl
invoc,fnct
ℓ.stat
value
ℓ.stat
w
τ π
=,stat,EXEC
var,π,τ
λ1,RETVAL,β
10
β ∈ {INT,REAL,RAT,LOG VAL}
decl
invoc
w
y ℓ.act
u ℓ.stat
v value
τ
short
ℓ.stat,END
x
→
decl
invoc
y ℓ.act
VAL,tov(τ)
v
value
v
τ
short
11
τ ∈ {int,rat,real,log,label}
decl
invoc
w
y ℓ.act
u ℓ.stat
v value
τ
long
ℓ.stat,END
x
→
decl
invoc
y ℓ.act
LVAL,tov
v (τ)
value
v
τ
long
12



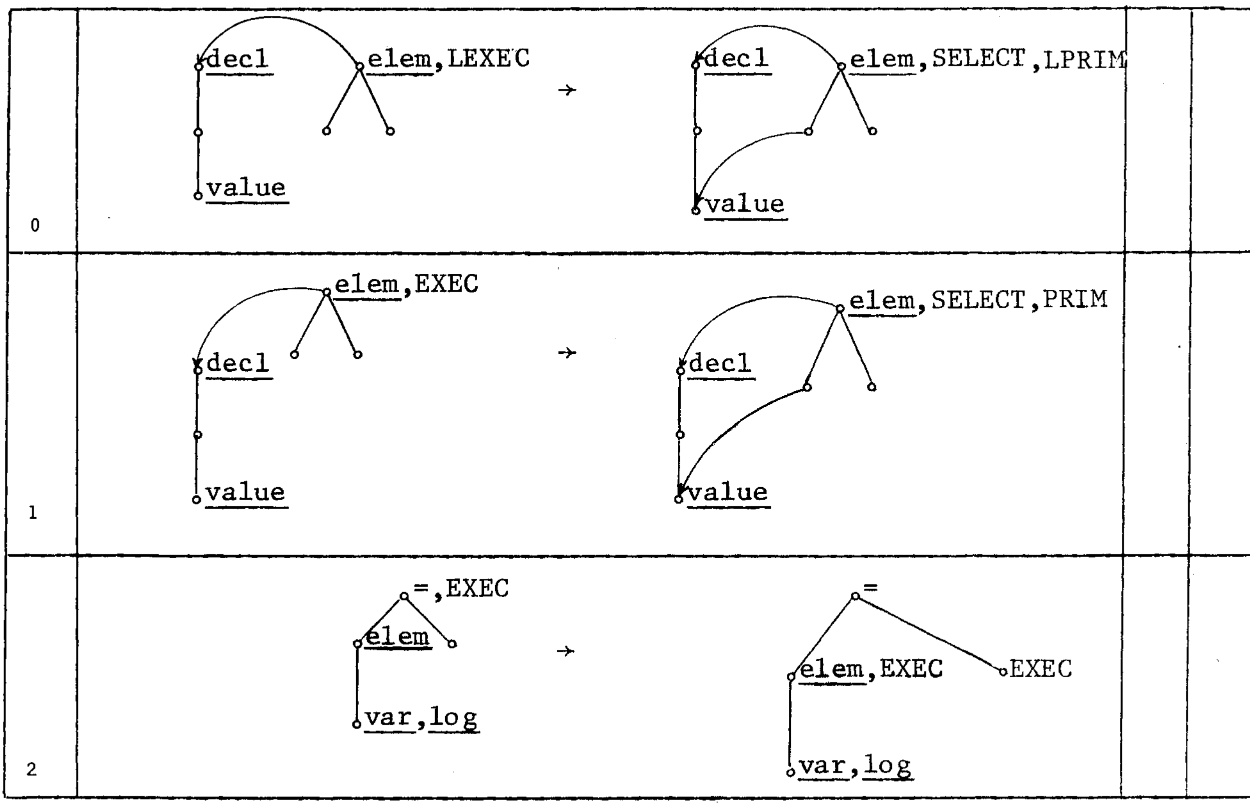
decl
invoc
w
ℓ.act
y
ℓ.stat
u
value
v
alg
short
ℓ.stat,END
x
→
decl
invoc
ℓ.act
y
VAL,ALG
v
value
v
alg
short
13
decl
invoc
w
ℓ.act
y
ℓ.stat
u
value
v
alg
long
ℓ.stat,END
→
decl
invoc
ℓ.act
y
VAL,ALG
v
value
v
alg
long
14
ℓ.stat
x
invoc,stat
w
ℓ.stat
x
y
ℓ.stat
u
VAL,RETVAL
stat
→
ℓ.stat
x
invoc
w
stat,goto
ℓ.stat
xy
ℓ.stat,END
u
stat
15

5. Element Accessing

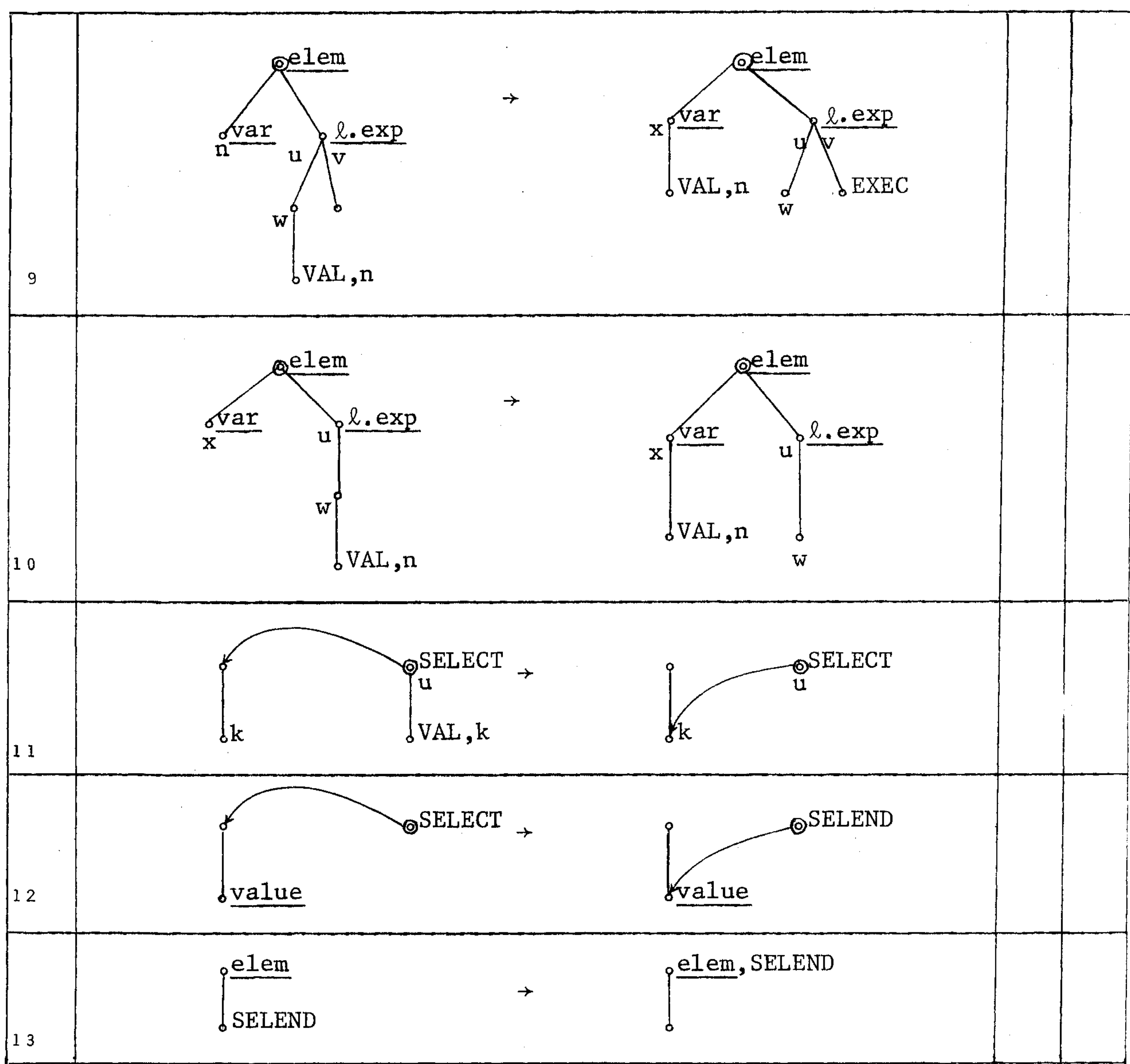
a) When we want to use an array of any type we must use it element by element. Two different usage of an element are possible. The first one is to use its value in an expression (0,1), the second one to assign to it a certain value (2-6). These two usages have in common a certain procedure which is to access an element of the array.

This selection is performed using the control word SELECT. We have to compute the subscripts (7-10) and locate the element in the value tree of the array (11,12).

b)



3	=,EXEC elem var,label → = elem,EXEC var,label	L_2	L_1
L_1 4	=,EXEC elem var,label → = elem,EXEC var,label		
L_2 4'	= elem,EXEC elem var,label var → elem,EXEC elem,EXEC var,label var		
5	=,EXEC elem REAL var → = elem,EXEC REAL,LEXEC var		
6	=,EXEC elem θ var → = elem,EXEC θ,.EXEC var θ ∈ {INT,RAT,ALG}		
7	elem,SELECT → elem EXEC		
8	ℓ.exp,EXEC u exp → ℓ.exp exp,EXEC		



6. Logical operations

a) 6.1 Logical assignment

Two cases to consider:

- simple variable as lefthand side (1,4)
- subscripted variable as lefthand side ($5.b_2$).

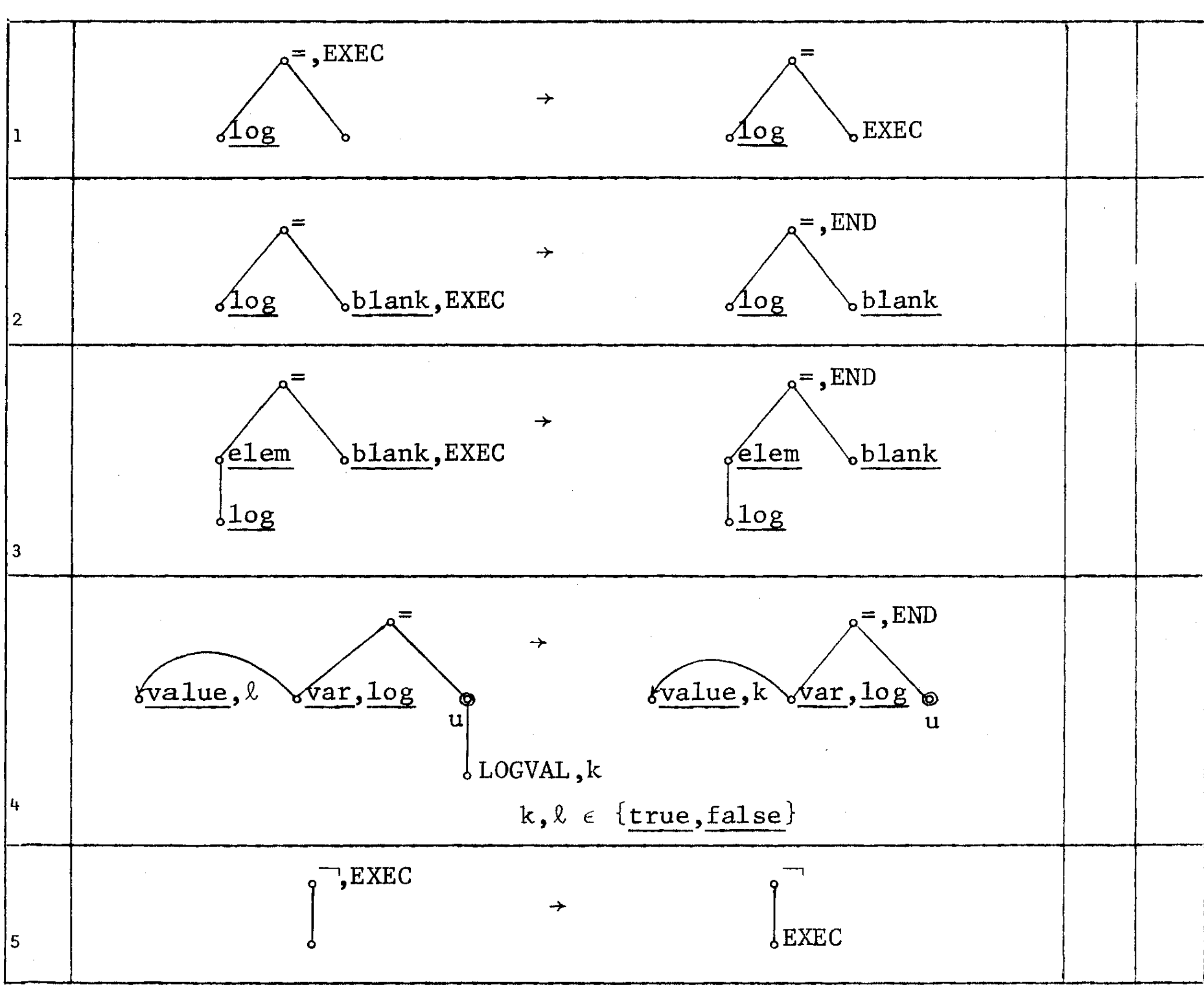
When the righthand side is blank, the assignment has no effect (2,3).

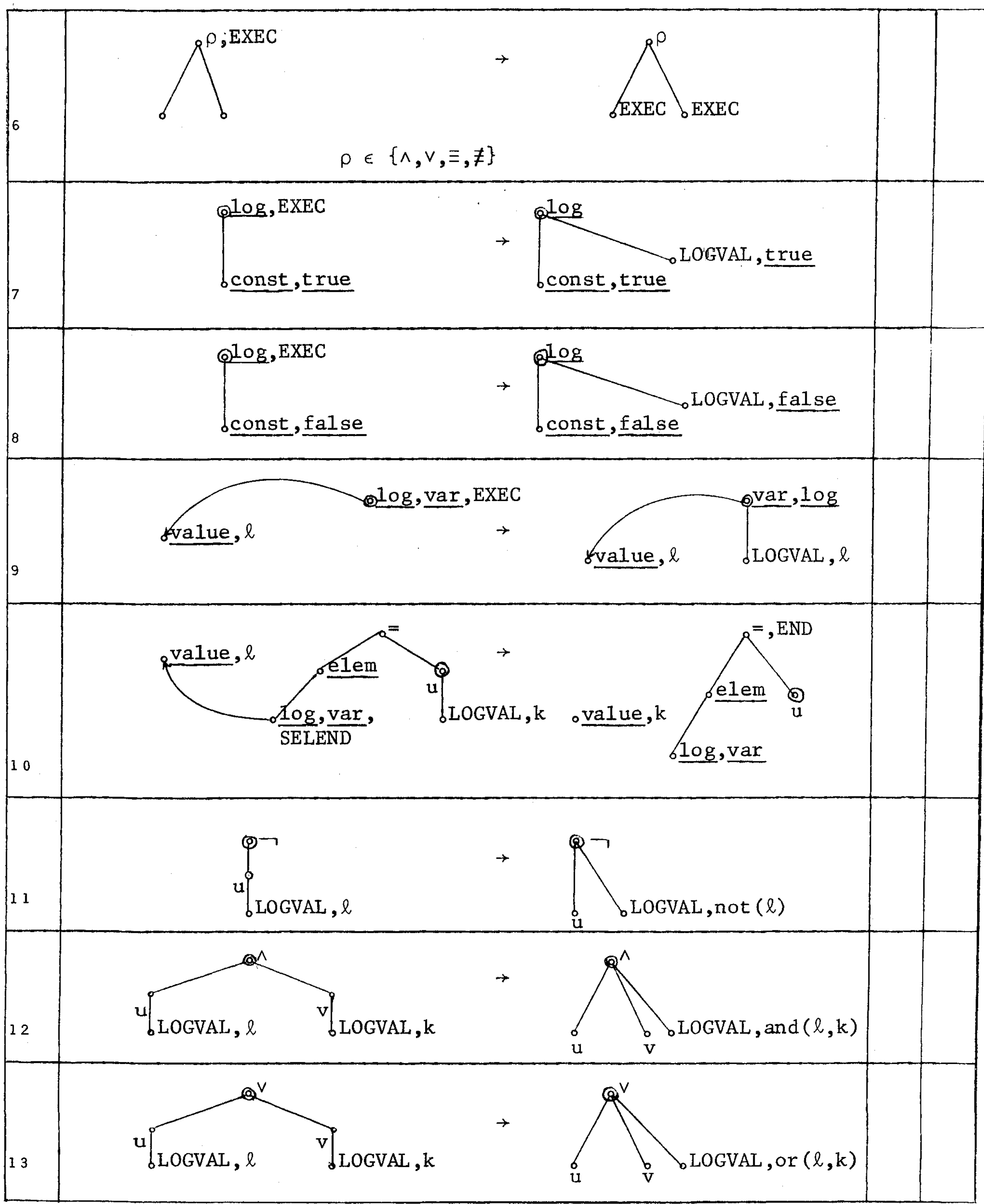
6.2 Logical expressions

A label parameter $\rho \in \{\wedge, \vee, \equiv, \not\equiv\}$ is used to make the presentation more compact (6).

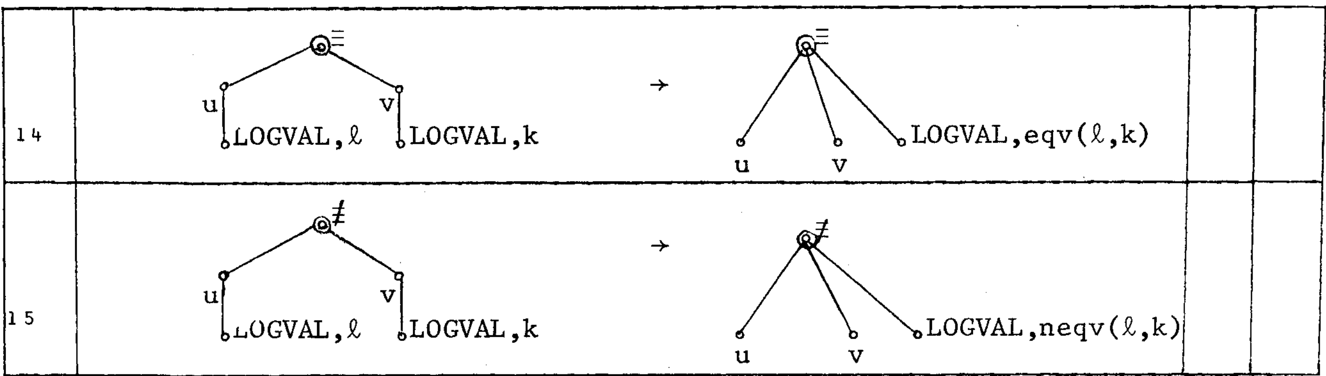
For the evaluation of a logical expression we fetch the value of constants and variables and when found we label it LOGVAL (7-10). Then we compute the result of a logical operation using five logical basic functions: not, and, or, eqv, neqv. Their meaning is well known (11-15).

b)





6
ρ,EXEC
→
ρ
EXEC
EXEC
ρ ∈ {∧,∨,≡,≢}
7
log,EXEC
const,true
→
log
LOGVAL,true
const,true
8
log,EXEC
const,false
→
log
LOGVAL,false
const,false
9
value,ℓ
log,var,EXEC
→
var,log
value,ℓ
LOGVAL,ℓ
10
value,ℓ
=
elem
log,var,
SELEND
u
LOGVAL,k
→
=,END
elem
u
value,k
log,var
11
¬
u
LOGVAL,ℓ
→
¬
u
LOGVAL,not(ℓ)
12
∧
u
LOGVAL,ℓ
v
LOGVAL,k
→
∧
u
v
LOGVAL,and(ℓ,k)
13
∨
u
LOGVAL,ℓ
v
LOGVAL,k
→
∨
u
v
LOGVAL,or(ℓ,k)

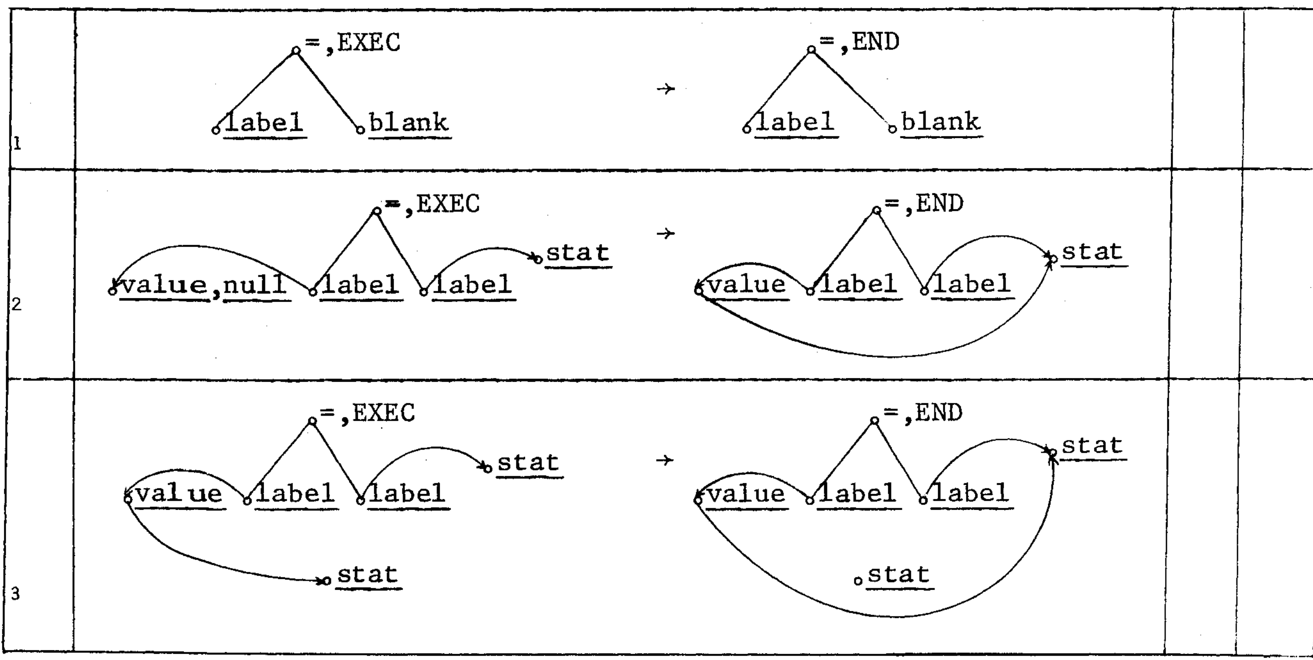


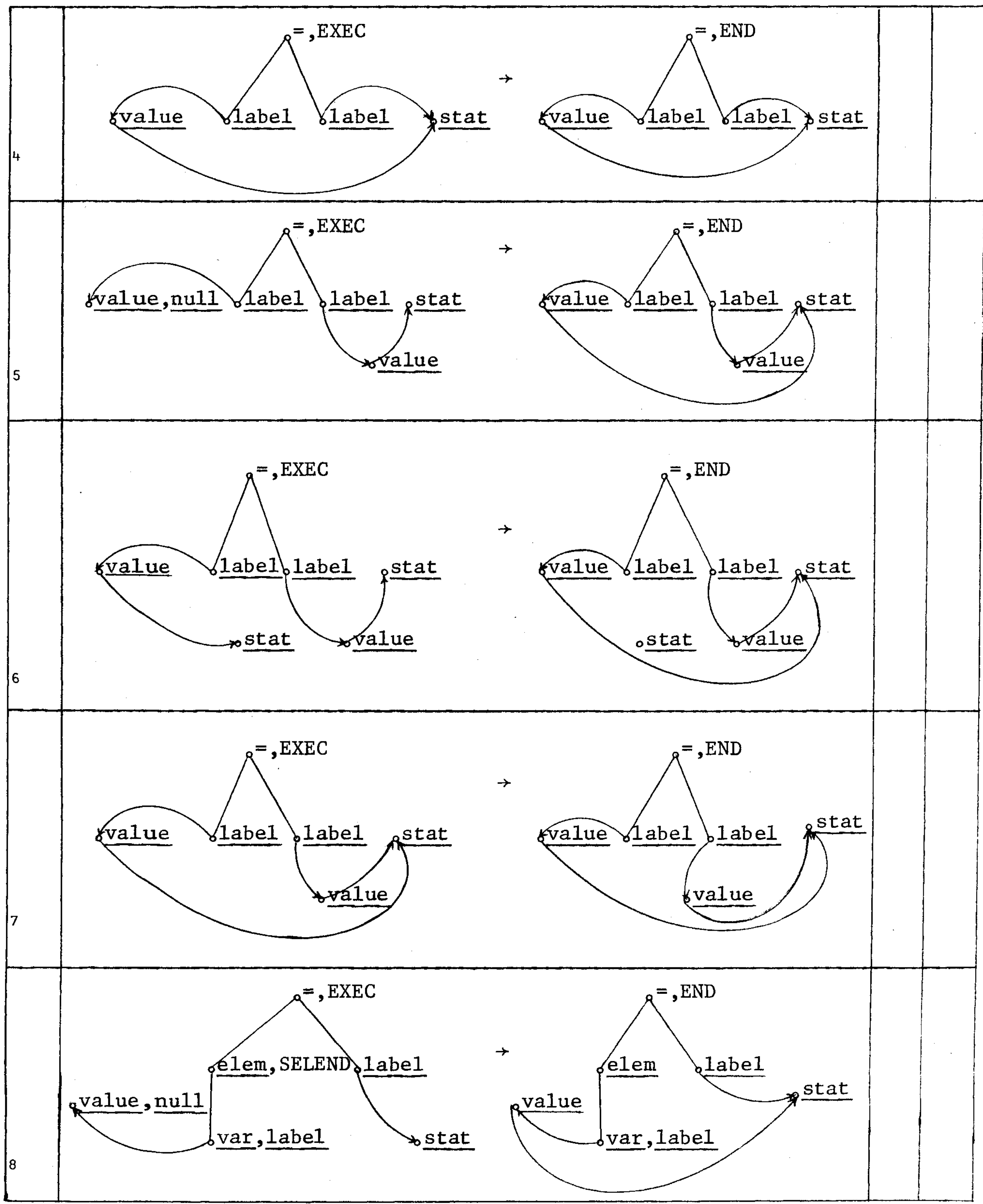
7. Label assignment

a) The label assignment should have as lefthand side a label variable that can be a simple variable or a subscripted variable element of an array. The righthand side can be a label variable (simple or subscripted) a label constant or a blank.

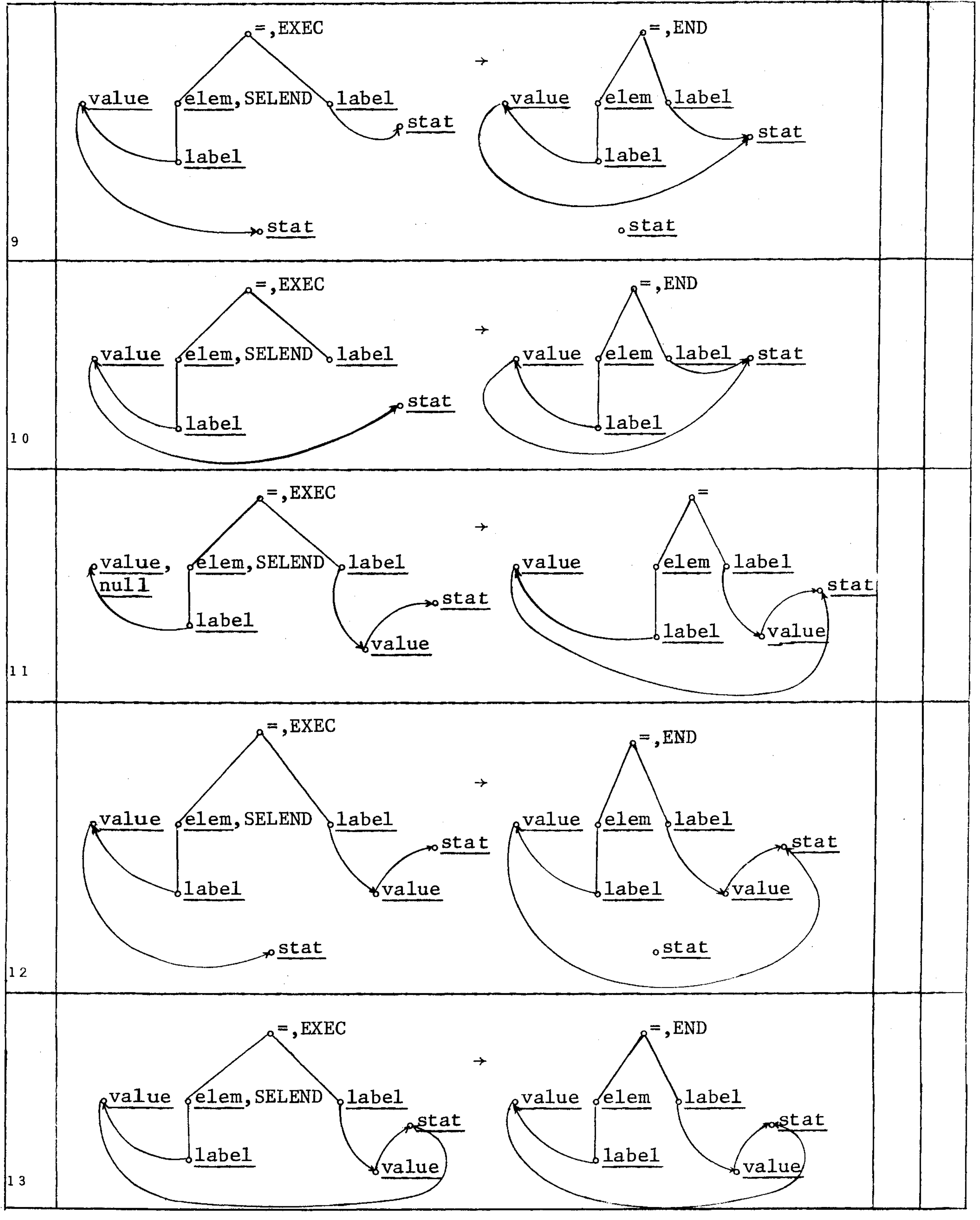
All the combinations should be considered one by one. Their meaning is obvious.

b)

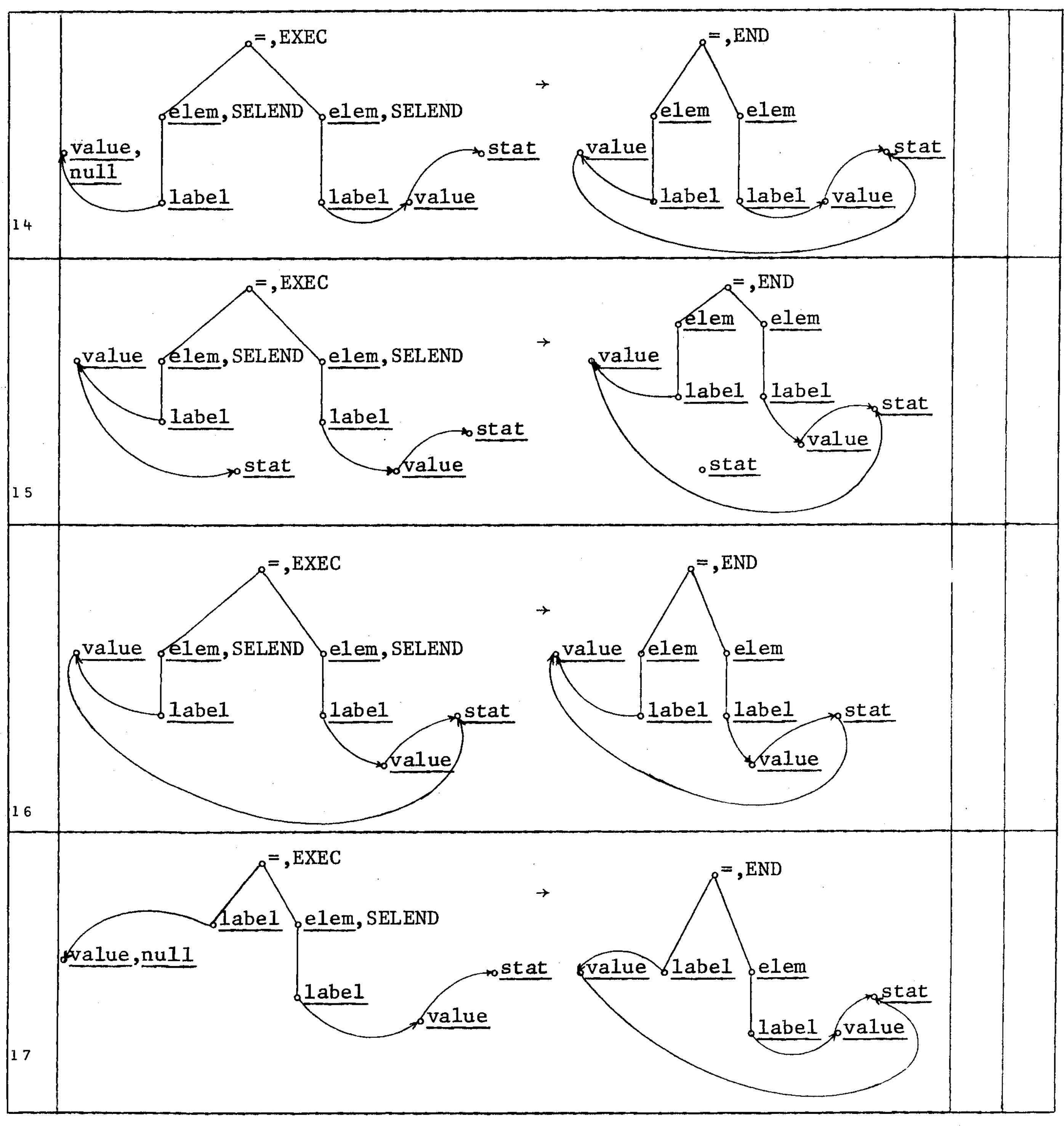




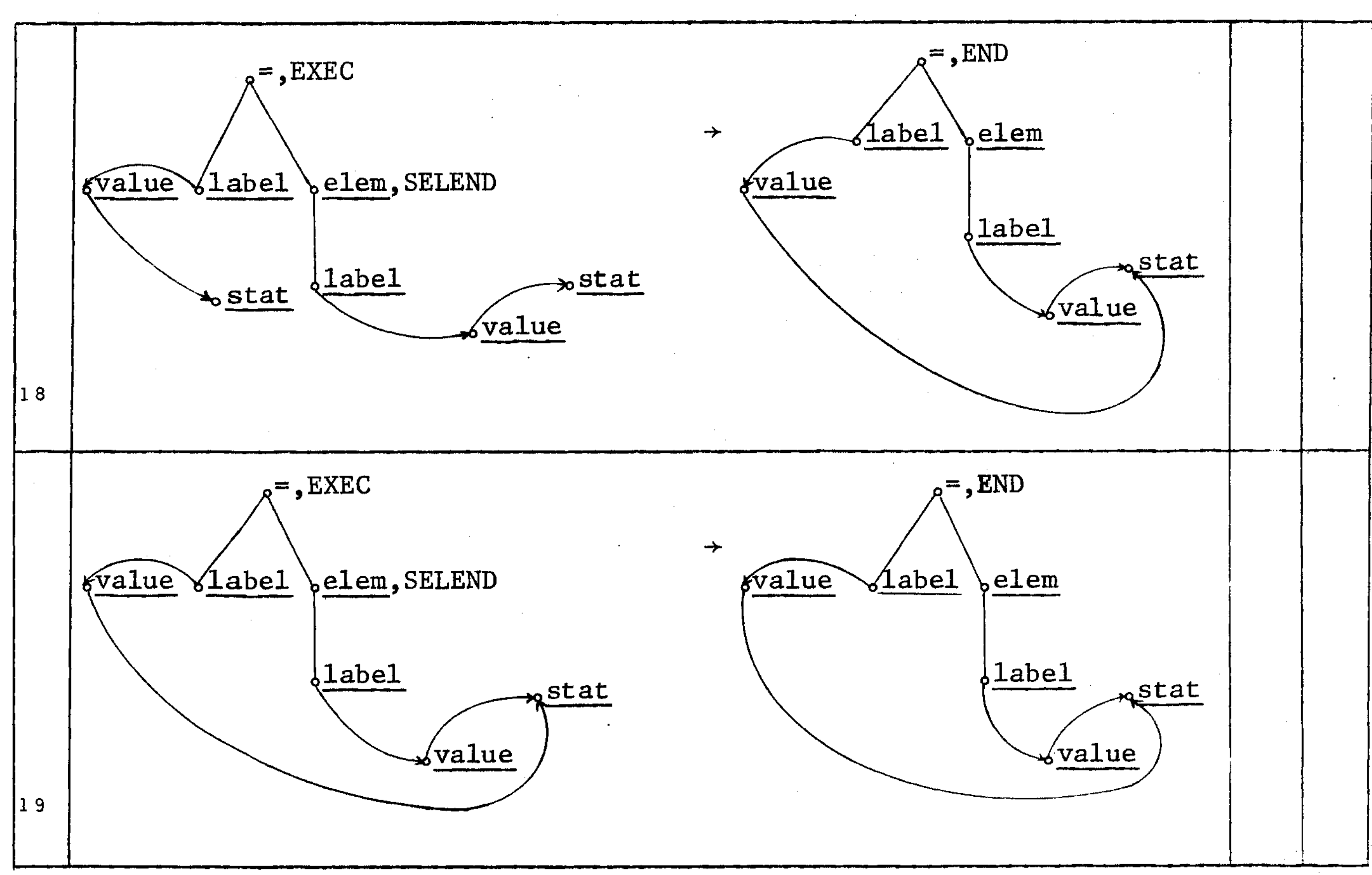
4
=,EXEC
value
label
label
stat
→
=,END
value
label
label
stat
5
=,EXEC
value,null
label
label
stat
value
→
=,END
value
label
label
stat
value
6
=,EXEC
value
label
label
stat
stat
value
→
=,END
value
label
label
stat
stat
value
7
=,EXEC
value
label
label
stat
value
→
=,END
value
label
label
stat
value
8
=,EXEC
elem,SELEND
label
value,null
var,label
stat
→
=,END
elem
label
stat
value
var,label



9
=,EXEC
value
elem,SELEND
label
stat
label
stat
→
=,END
value
elem
label
stat
label
stat
10
=,EXEC
value
elem,SELEND
label
label
stat
→
=,END
value
elem
label
stat
label
11
=,EXEC
value,
null
elem,SELEND
label
label
stat
value
→
=
value
elem
label
stat
label
value
12
=,EXEC
value
elem,SELEND
label
stat
label
value
stat
→
=,END
value
elem
label
stat
label
value
stat
13
=,EXEC
value
elem,SELEND
label
stat
label
value
→
=,END
value
elem
label
stat
label
value



14
=,EXEC
elem,SELEND
elem,SELEND
value,
null
label
label
value
stat
→
=,END
elem
elem
value
label
label
value
stat
15
=,EXEC
value
elem,SELEND
elem,SELEND
label
label
stat
stat
value
→
=,END
elem
elem
value
label
label
stat
value
stat
16
=,EXEC
value
elem,SELEND
elem,SELEND
label
label
stat
value
→
=,END
value
elem
elem
label
label
stat
value
17
=,EXEC
label
elem,SELEND
value,null
label
stat
value
→
=,END
value
label
elem
stat
label
value



8. The arithmetic assignment statement

8.1 Conversion of precision

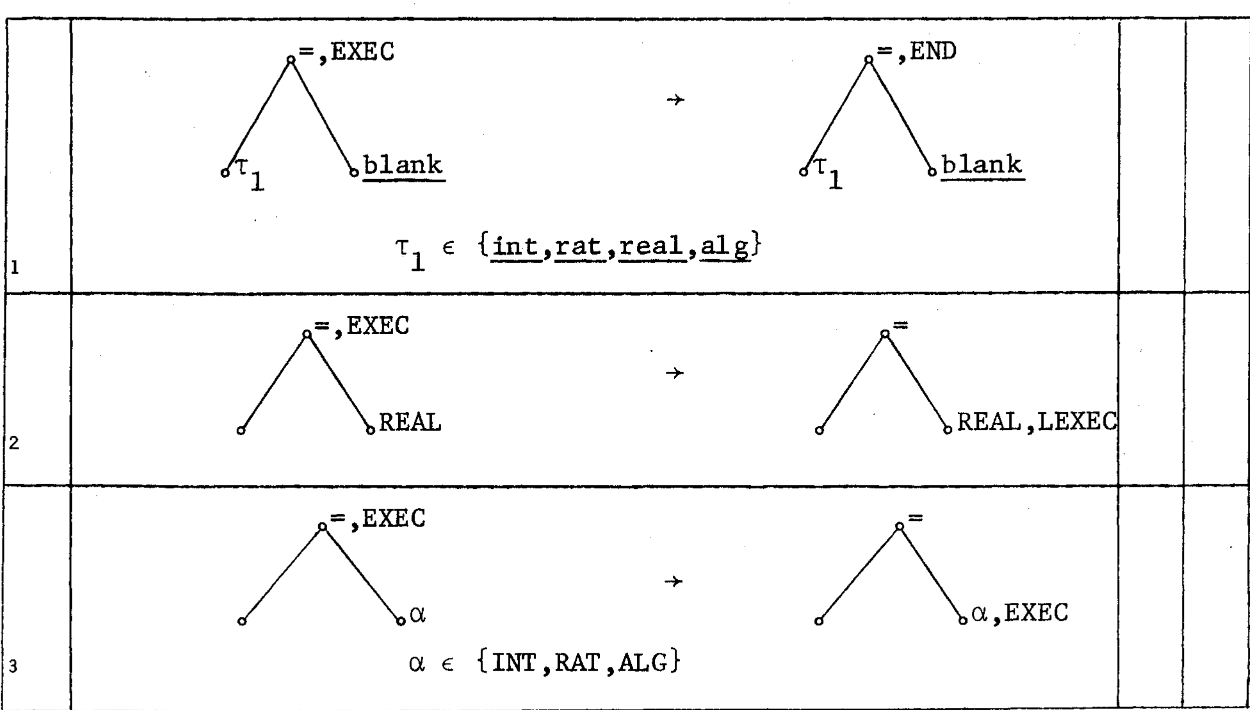
a) The arithmetic assignment statement has an arithmetic expression as righthand side. It should have been labeled REAL, INT, RAT or ALG while verifying the compatibility of types in the Syntax Description. If this expression is REAL then the computations will be made in long precision (LEXEC). Otherwise, the computations will be started in short precision (EXEC) (2, 3).

When the value of the expression is obtained and is short (VAL), while the lefthand side is long, then the value of the expression is converted to long precision (LVAL) (4).

If the value of the righthand side is a long integer while the lefthand side is short, then the value of the righthand side is converted to a short integer if this is possible. The conversion is done by the basic function intsh defined in the formal part below (5).

Similar situations to this one but with a real, rational, or algebraic expression as righthand side are considered. A function realsh, that truncates the mantissa of a real number, is used. For the rational conversion, the function intsh (on integers) is used. But for the algebraic conversion a macro operation denoted alsh is used to truncate all the coefficients (integer or rational) of the algebraic if this is possible (6,7,8).

b)



4	= long VAL	→	= long VAL		
5	= short INT,LVAL,χ	→	= short INT,VAL,intsh(χ)		
6	= short LVAL,χ,REAL	→	= REAL,VAL,realsh(χ)		
7	= short /,RAT,LVAL χ_1 χ_2	→	= /,RAT,VAL intsh(χ_1) intsh(χ_2)		
8	= short ALG,LVAL v	→	= ALG,VAL alsh(v)		

The function intsh is defined by

$$\text{intsh} : I_s \cup \bar{I}_s \rightarrow I_s \cup \bar{I}_s$$

$$x \rightarrow x$$

This function is not defined for $x \in I_\ell \cup \bar{I}_\ell$.

The function realsh : $R_\ell \cup \bar{R}_\ell \rightarrow R_s \cup \bar{R}_s$ truncates the real number to make it short. $\bar{R}_\ell$ and $\bar{R}_s$ are respectively the set of long negative real numbers and the set of short negative real numbers.

8.2 The conversion of type

a) When in an assignment the types of the lefthand side and of the righthand side differ there is some rule for conversion that make it possible in most cases.

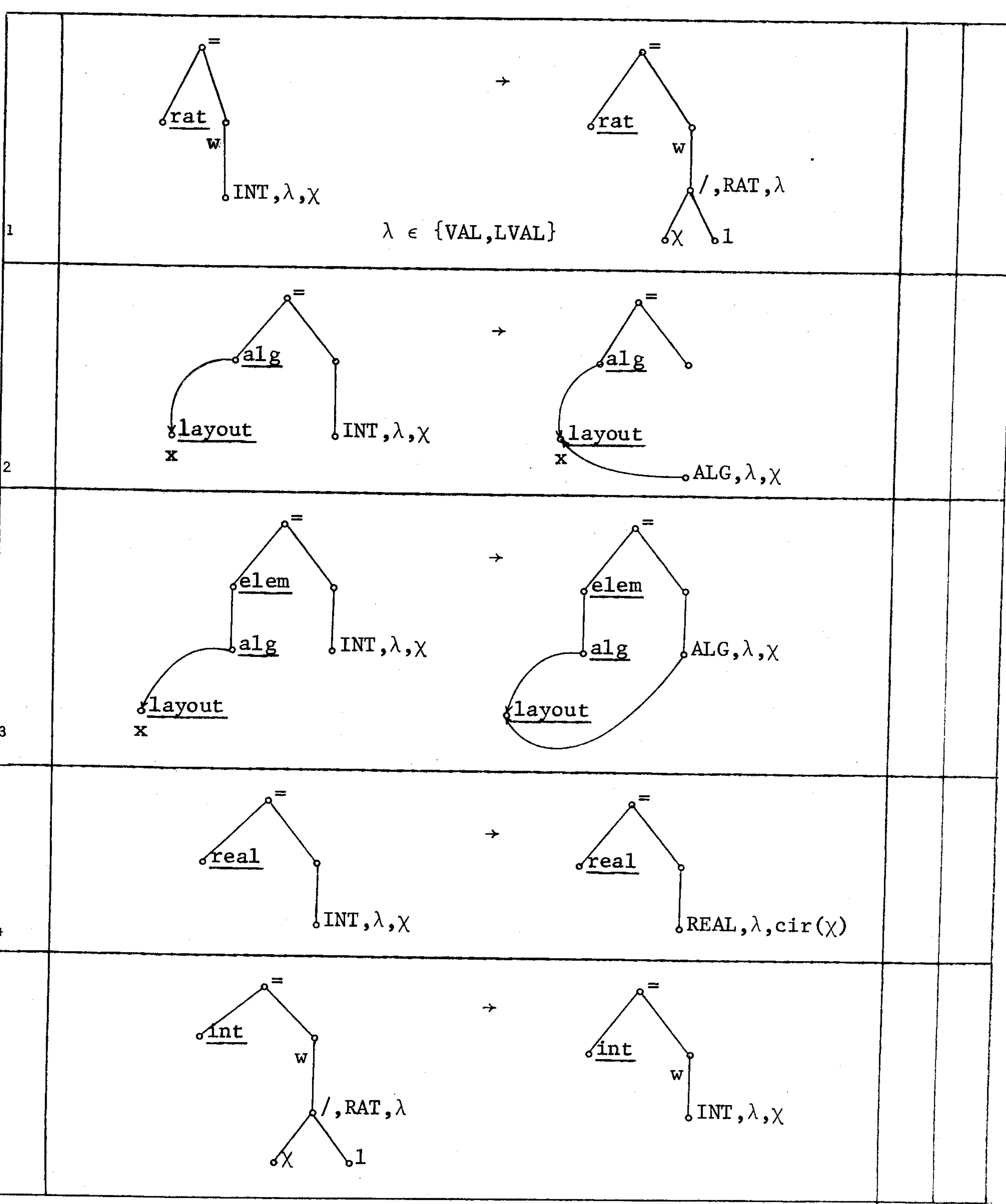
The conversion from INT to RAT, ALG or REAL is always possible (1-4). The function cir (conversion from integer to real) is used and is defined by cir : $Z \rightarrow R$, $\text{cir}(z) = z$.

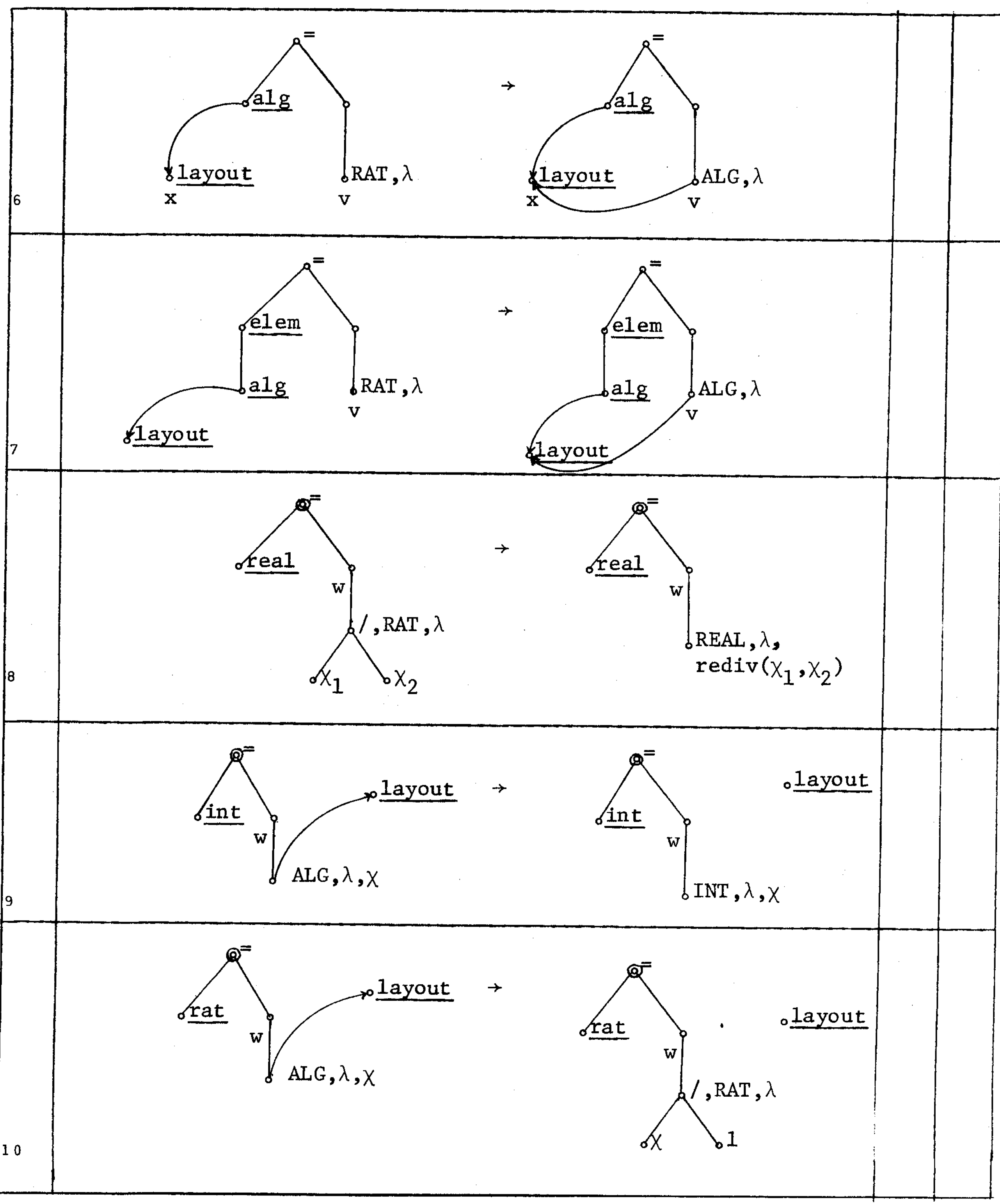
The conversion from RAT to ALG or REAL is always possible, but that from RAT to INT is only possible if the denominator of the rational number is one (5-8). The conversion to real makes use of the function rediv (real division) that is defined by rediv : $Z \times Z \rightarrow R$, rediv $(z_1, z_2) = z_1/z_2$.

The conversion from ALG to INT, RAT or REAL is possible when the value is numeric and for conversion to INT it should have 1 for denominator, if it is a rational value (9-13).

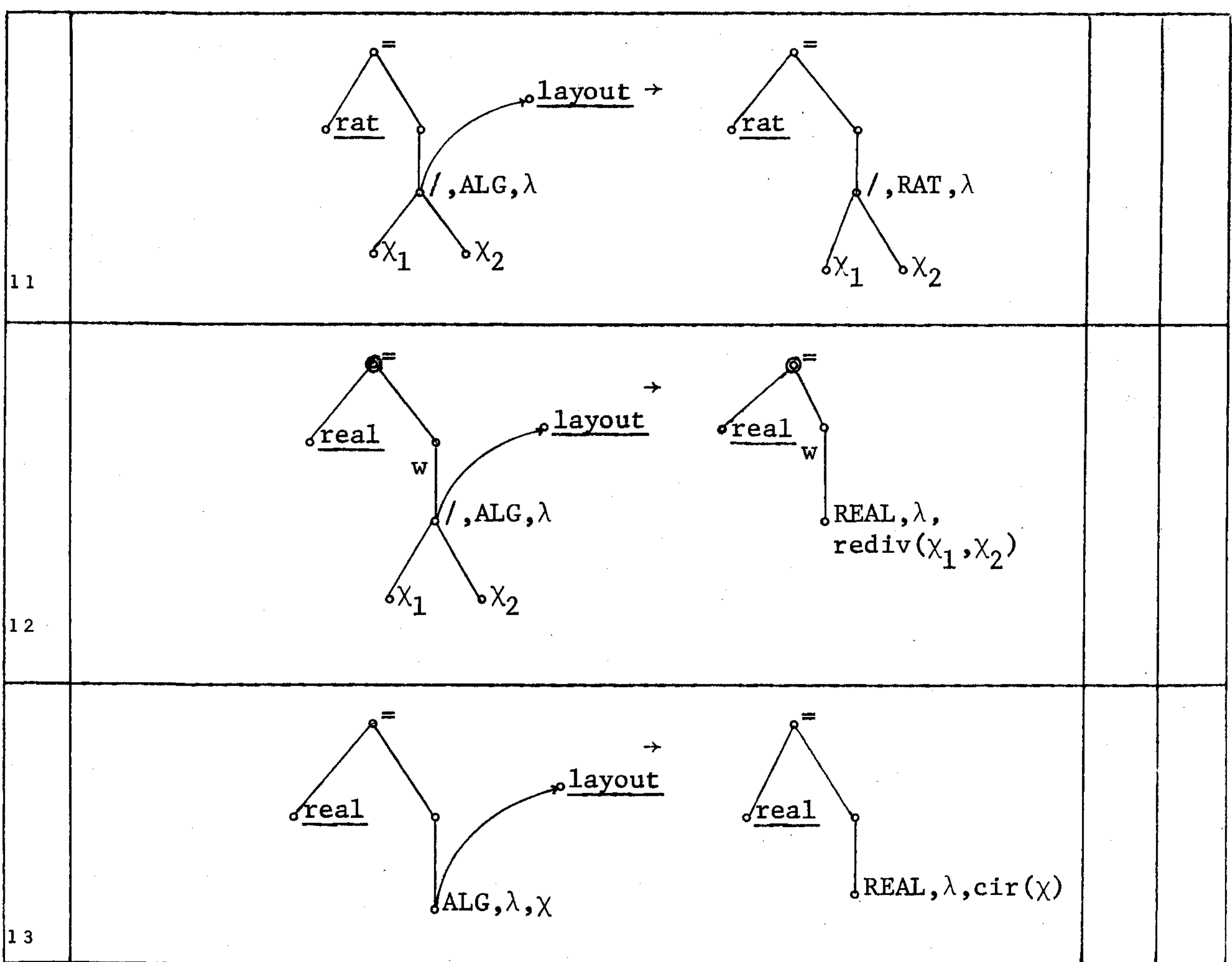
All these cases are common to simple and subscripted variables. Moreover, when an operand is algebraic the layout should be transmitted or deleted depending on the context.

b)





6
=
alg
layout
x
RAT,λ
v
→
=
alg
layout
x
ALG,λ
v
7
=
elem
alg
layout
RAT,λ
v
→
=
elem
alg
ALG,λ
v
layout
8
=
real
w
/,RAT,λ
χ1
χ2
→
=
real
w
REAL,λ,
rediv(χ1,χ2)
9
=
int
w
ALG,λ,χ
layout
→
=
int
w
INT,λ,χ
layout
10
=
rat
w
ALG,λ,χ
layout
→
=
rat
w
/,RAT,λ
χ
1
layout



8.3 The assignment of a value

The assignment of a value of any type is done when both sides of the assignment statement have the same precision and the same type.

However, for the particular case of an algebraic we must check that the layout of the lefthand side is respected. This is done in two steps:

- The first step uses a macro, check, which when applied to an algebraic value gives a list of the indeterminates used in this algebraic value with their highest exponent. This list is in structure analog to a layout (1).

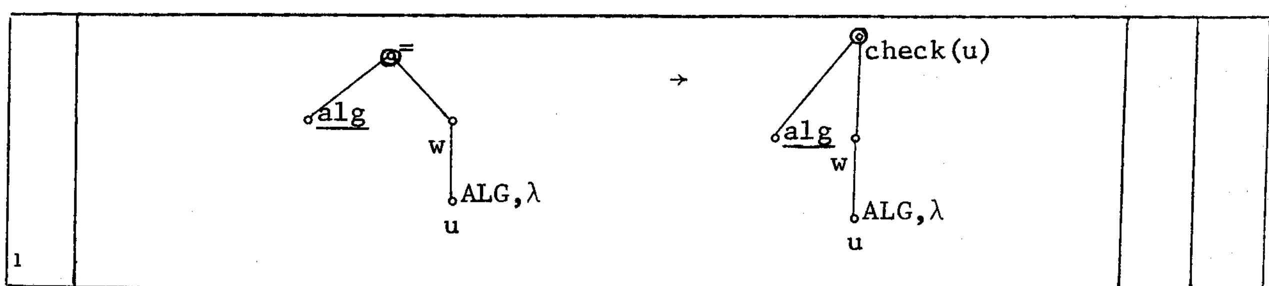
– The second step makes use of a logical function, icomp (integer comparison) which is defined later on formally. This basic function when given as arguments a relation symbol and two integers, yields as value <u>true</u> if the relation holds between the two integers, <u>false</u> otherwise. So using this function we can check that the highest exponents are respected for each indeterminate. Productions (2,3) perform this step, and show that it is an error to have an indeterminate in the algebraic value that violates the layout, i.e. that is not in the layout of the lefthand side variable.

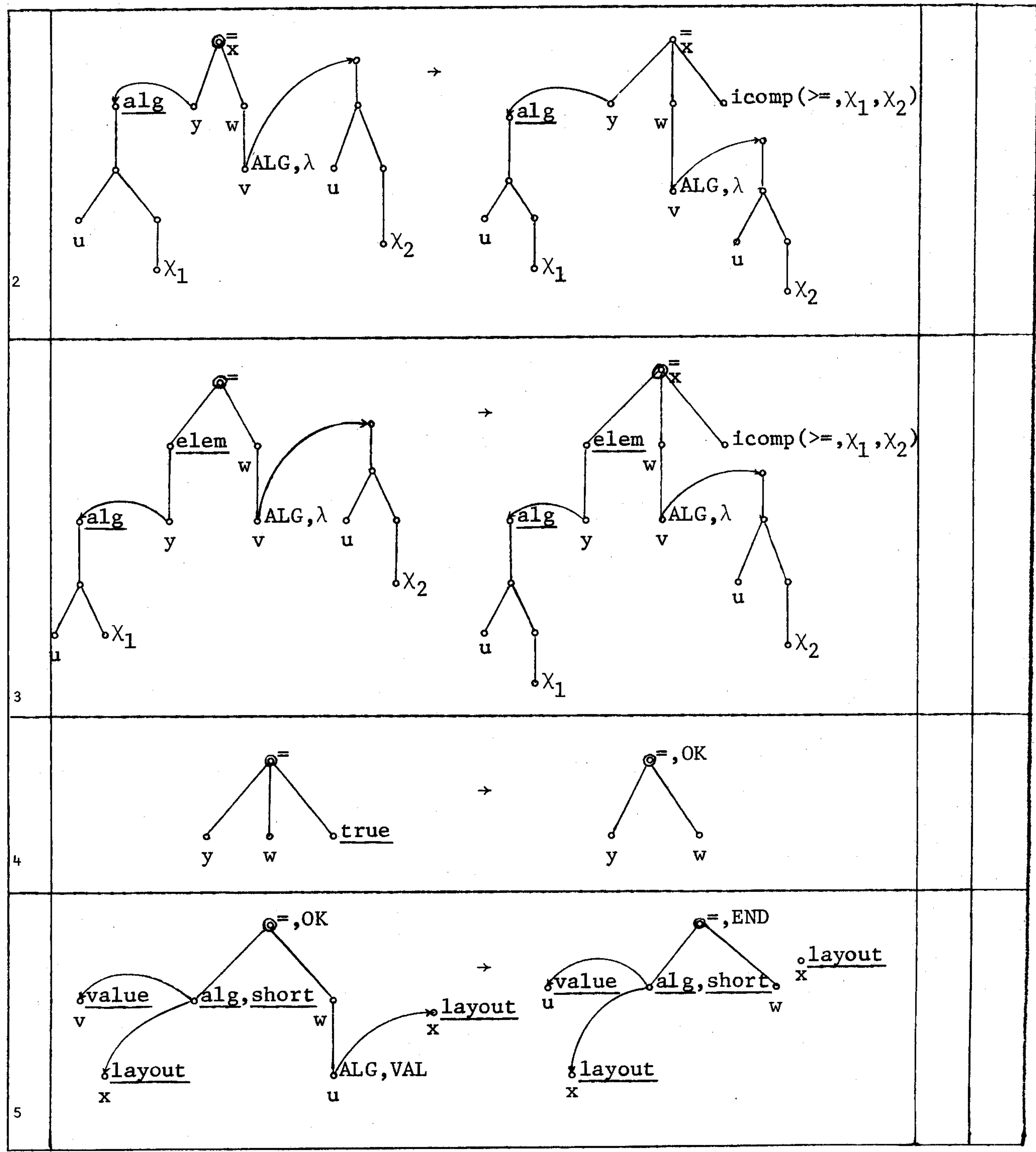
The label OK is attached after the first indeterminate is checked but in fact we must continue applying production (4) as long as there is pairs (indeterminate: highest exponent) hanging from the assignment tree and before we can apply the production that assigns the value to the algebraic variable.

To assign the righthand side value to the lefthand side variable the layout of both sides should be the same (5).

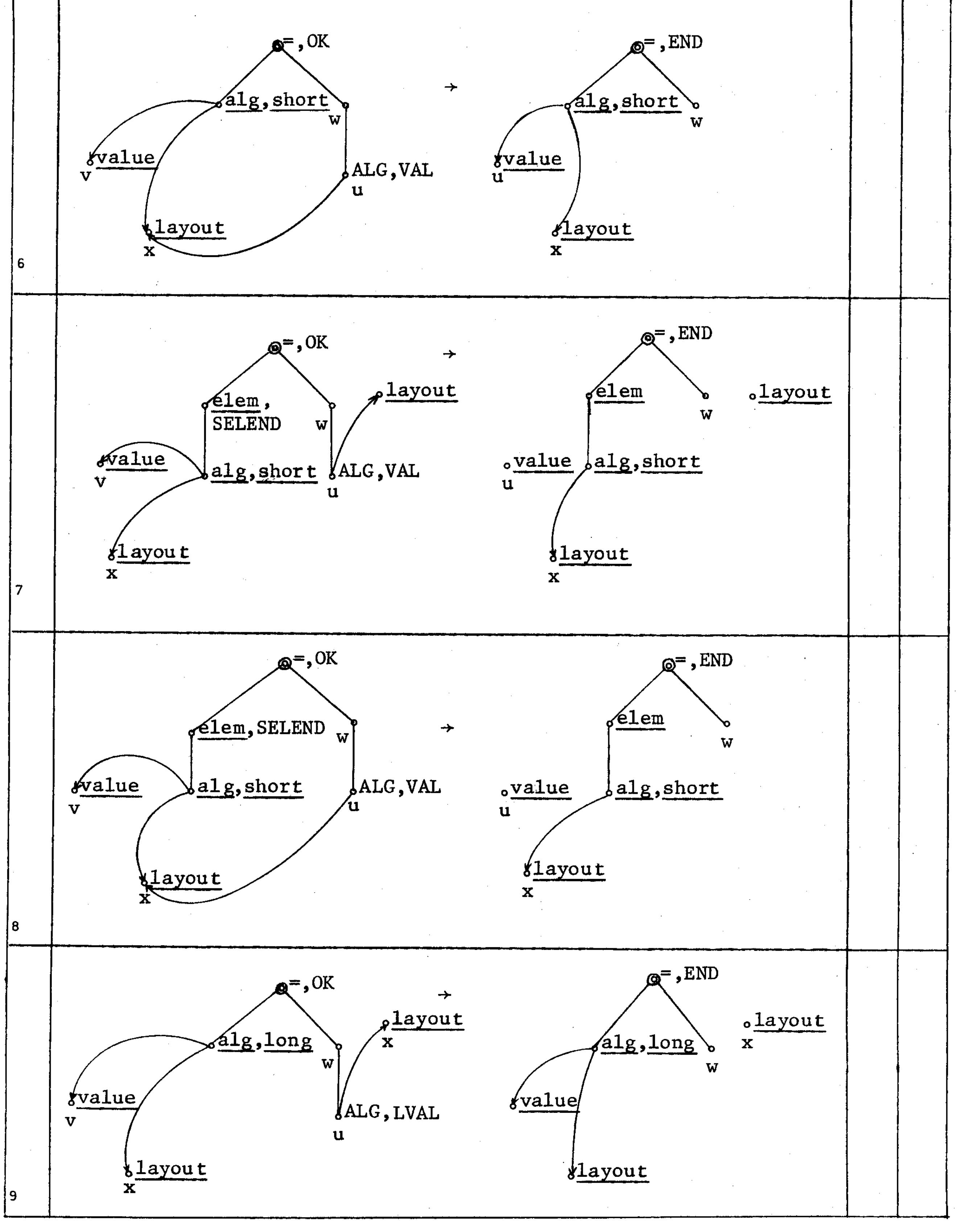
For the other types the assignment of value is easy to understand. There is also a compulsory distinction to make between simple and subscripted variables: the pointer to the value of a subscripted variable should be deleted after the assignment is performed. But <u>elem</u>-node keeps pointing to <u>decl</u>-node of the array.

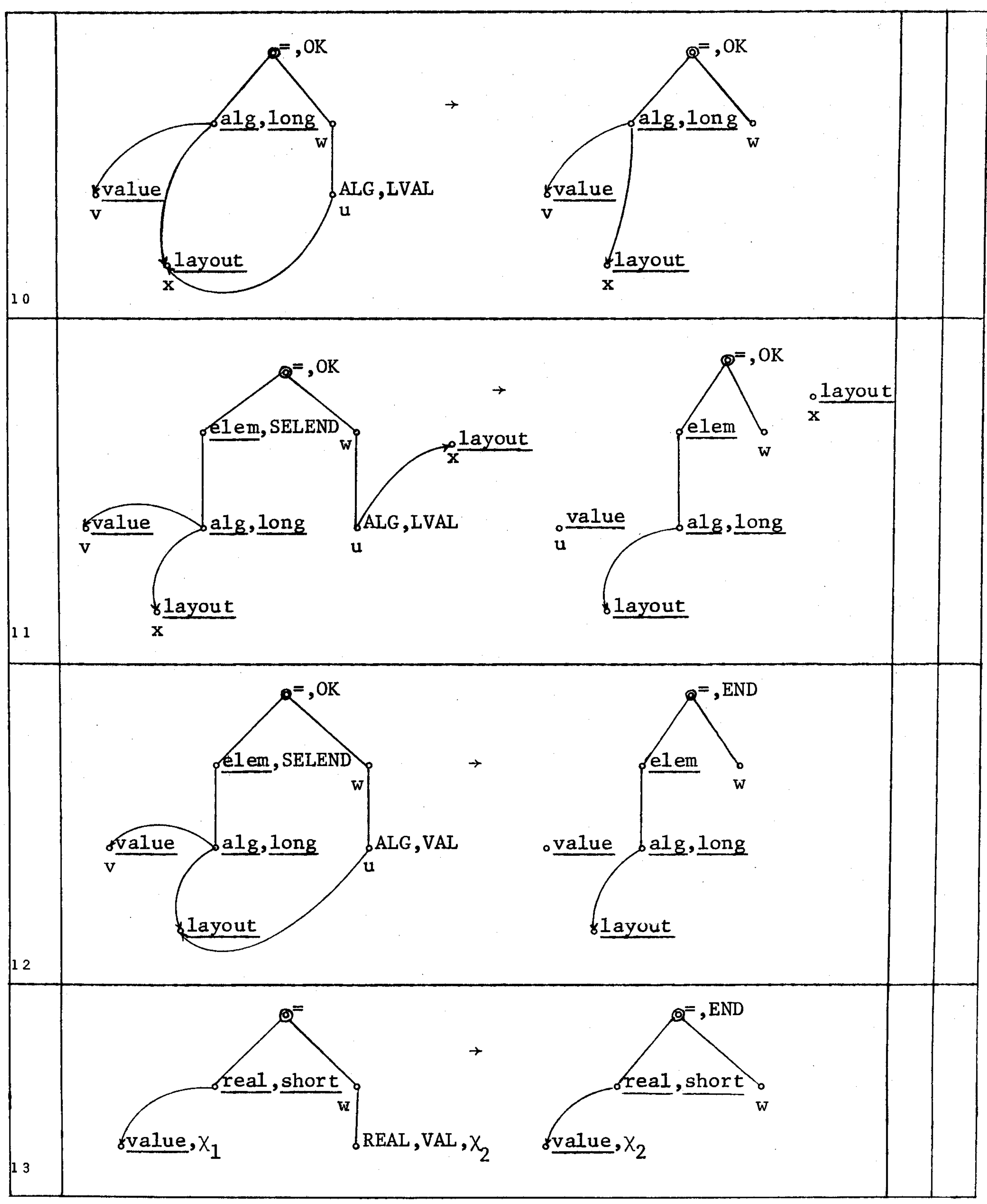
b)



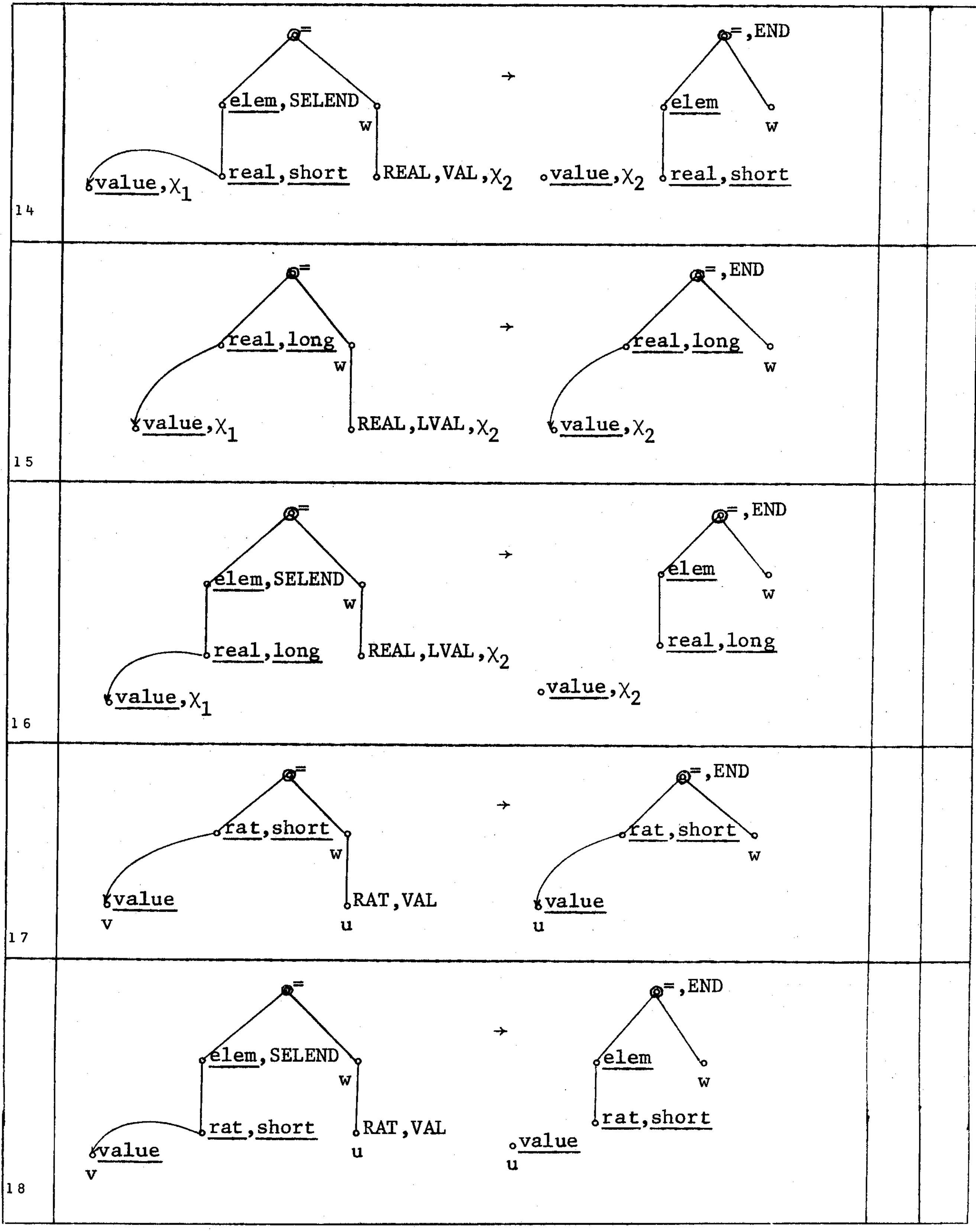


2
alg
y
w
ALG,λ
v
u
u
χ_1
χ_2
→
x
icomp(>=,χ_1,χ_2)
3
elem
4
true
=,OK
5
value
alg,short
layout
ALG,VAL
=,END

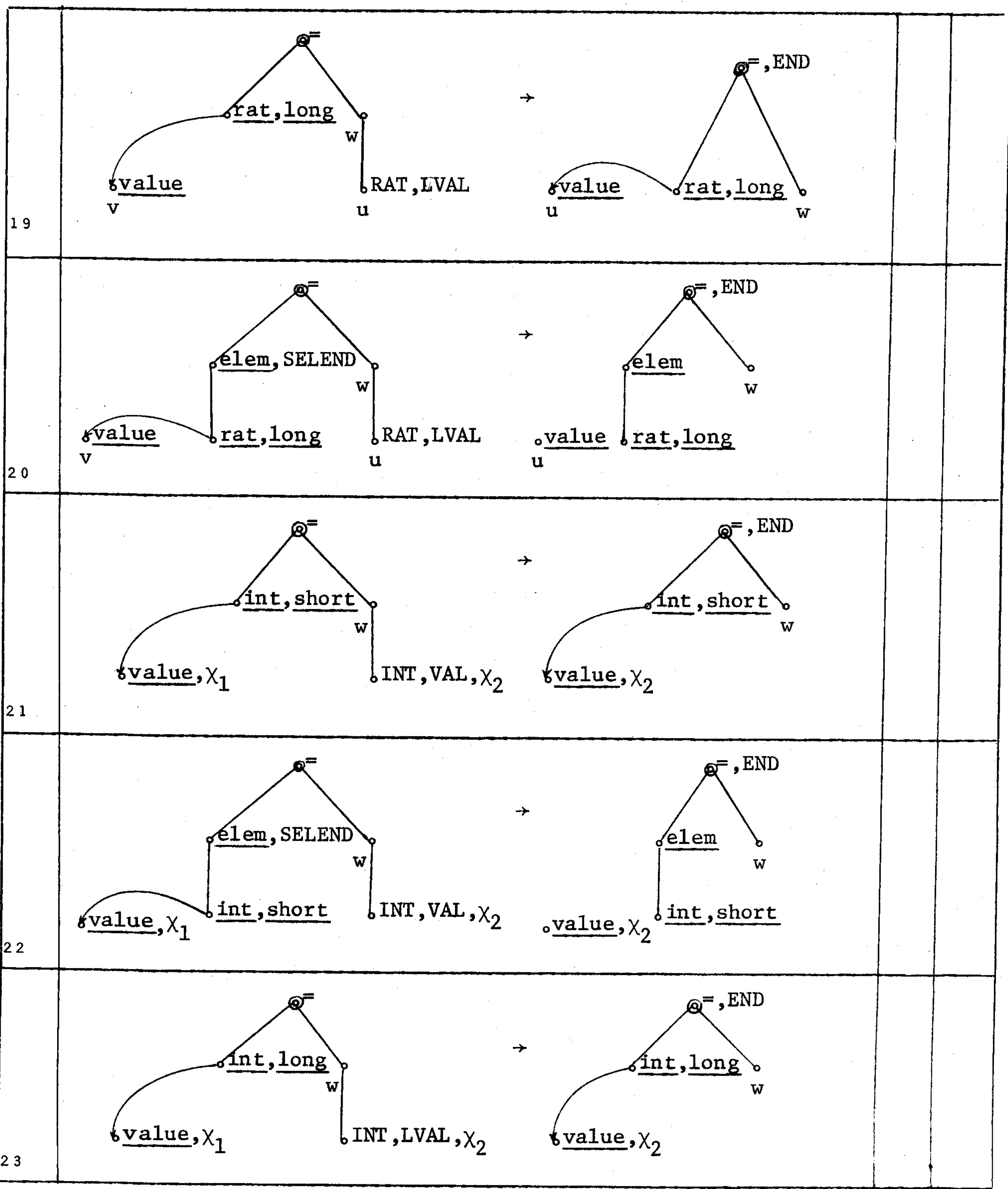

6
=,OK
alg,short
w
value
v
ALG,VAL
u
layout
x
→
=,END
alg,short
w
value
u
layout
x
7
=,OK
elem,
SELEND
w
layout
value
v
alg,short
ALG,VAL
u
layout
x
→
=,END
elem
w
layout
value
u
alg,short
layout
x
8
=,OK
elem,SELEND
w
value
v
alg,short
ALG,VAL
u
layout
x
→
=,END
elem
w
value
u
alg,short
layout
x
9
=,OK
alg,long
w
layout
x
value
v
ALG,LVAL
u
layout
x
→
=,END
alg,long
w
layout
x
value
layout



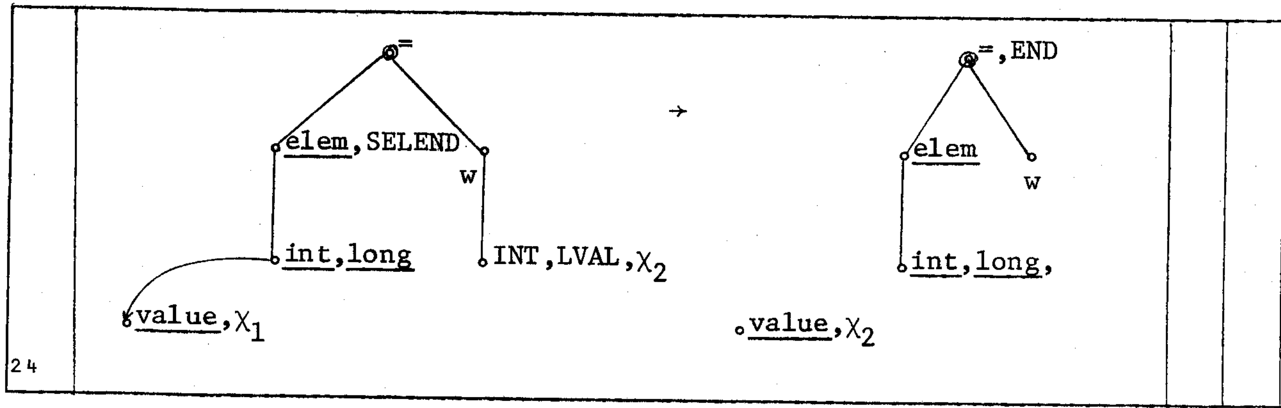
10
=,OK
alg,long
w
value
v
ALG,LVAL
u
layout
x
→
=,OK
alg,long
w
value
v
layout
x
11
=,OK
elem,SELEND
w
layout
x
value
v
alg,long
ALG,LVAL
u
layout
x
→
=,OK
elem
w
layout
x
value
u
alg,long
layout
12
=,OK
elem,SELEND
w
value
v
alg,long
ALG,VAL
u
layout
→
=,END
elem
w
value
alg,long
layout
13
=
real,short
w
value,χ1
REAL,VAL,χ2
→
=,END
real,short
w
value,χ2



14
=
elem, SELEND
w
real, short
value, χ_1
REAL, VAL, χ_2
→
=, END
elem
w
value, χ_2
real, short
15
=
real, long
w
value, χ_1
REAL, LVAL, χ_2
→
=, END
real, long
w
value, χ_2
16
=
elem, SELEND
w
real, long
REAL, LVAL, χ_2
value, χ_1
→
=, END
elem
w
real, long
value, χ_2
17
=
rat, short
w
value
v
RAT, VAL
u
→
=, END
rat, short
w
value
u
18
=
elem, SELEND
w
rat, short
RAT, VAL
u
value
v
→
=, END
elem
w
rat, short
value
u



=
rat,long
w
value
v
RAT,LVAL
u
→
=,END
value
u
rat,long
w
19
=
elem,SELEND
w
value
v
rat,long
RAT,LVAL
u
→
=,END
elem
w
value
u
rat,long
20
=
int,short
w
value,χ_1
INT,VAL,χ_2
→
=,END
int,short
w
value,χ_2
21
=
elem,SELEND
w
value,χ_1
int,short
INT,VAL,χ_2
→
=,END
elem
w
value,χ_2
int,short
22
=
int,long
w
value,χ_1
INT,LVAL,χ_2
→
=,END
int,long
w
value,χ_2
23



9. Arithmetic expressions

9.1 Evaluation of Constants and variables

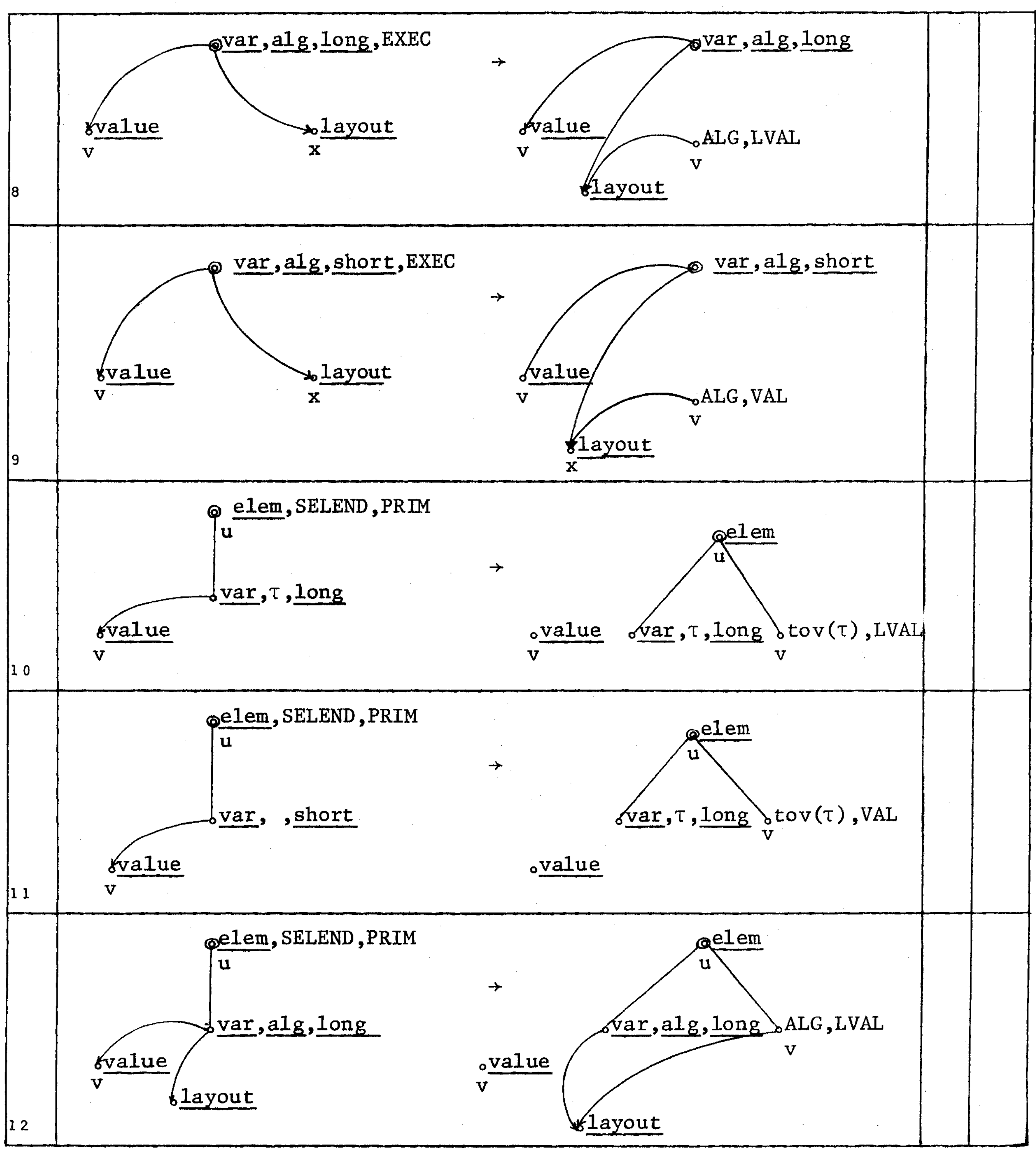
a) After we enter an expression, the evaluation is performed in parallel and naturally bottom-up (1,2). So we have to look for constants and variables and evaluate them. To specify the type of the value obtained we use a function tov (type of value) to make the presentation more compact. The two cases of long and short precision are considered separately as well as the cases of simple and subscripted variables (3-17). It should be noted that when the execution is done in short precision (EXEC), and we encounter a long variable or constant then the execution is resumed in long precision. But when the execution is in long precision (LEXEC) and a short variable or constant is encountered, the execution continue in long precision.

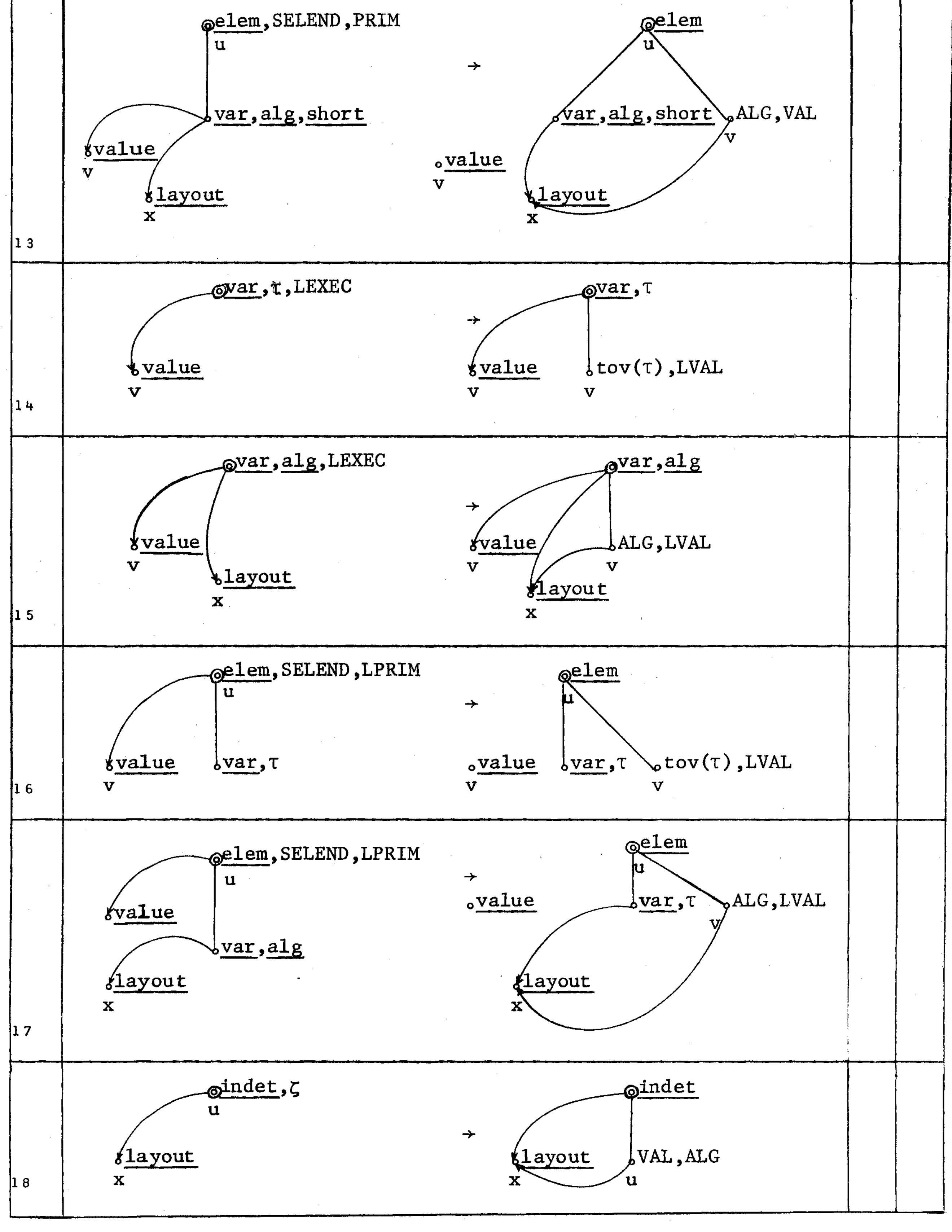
A label parameter ζ is used whose domain is {EXEC,LEXEC}. The label parameter η belongs to the domain $\{+,-,*,/,\uparrow\} \cup \{>,>=,<,<=,<>,==\}$.

For the algebraic variables or indeterminates the layout associated with the value of this variable or indeterminates is the layout of the variable itself (8,9). Indeterminates have for value their own name and naturally are considered as algebraic values(18).

b)

1	Θ, ζ (child node)	→	Θ ζ ζ ∈ {EXEC,LEXEC}		
2	η,ζ (two child nodes)	→	η ζ ζ η ∈ {+,-,*,/,↑} ∪ {>,>=,<,<=,<>,==}		
3	τ,short,EXEC const,χ	→	τ,short const,χ tov(τ),VAL,χ		
4	τ,long,EXEC const,χ	→	τ,long const,χ tov(τ),LVAL,ζ		
5	τ,LEXEC const,χ	→	τ const,χ tov(τ),LVAL,χ		
6	τ,long,var,EXEC value v	→	var,τ,long value tov(τ),LVAL v v τ ∈ {int,rat,real,log,label}		
7	var,τ,short,EXEC value v	→	var,τ,short value tov(τ),VAL v v		


var,alg,long,EXEC
value
v
layout
x
→
var,alg,long
value
v
ALG,LVAL
v
layout
8
var,alg,short,EXEC
value
v
layout
x
→
var,alg,short
value
v
ALG,VAL
v
layout
x
9
elem,SELEND,PRIM
u
var,τ,long
value
v
→
elem
u
value
v
var,τ,long
tov(τ),LVAL
v
10
elem,SELEND,PRIM
u
var, ,short
value
v
→
elem
u
var,τ,long
tov(τ),VAL
v
value
11
elem,SELEND,PRIM
u
var,alg,long
value
v
layout
→
elem
u
var,alg,long
ALG,LVAL
v
value
v
layout
12


13
elem,SELEND,PRIM
u
var,alg,short
value
v
layout
x
→
elem
u
var,alg,short
ALG,VAL
v
value
v
layout
x
14
var,τ,LEXEC
value
v
→
var,τ
value
v
tov(τ),LVAL
v
15
var,alg,LEXEC
value
v
layout
x
→
var,alg
value
v
ALG,LVAL
v
layout
x
16
elem,SELEND,LPRIM
u
value
v
var,τ
→
elem
u
value
v
var,τ
tov(τ),LVAL
v
17
elem,SELEND,LPRIM
u
value
var,alg
layout
x
→
elem
u
value
var,τ
ALG,LVAL
v
layout
x
18
indet,ζ
u
layout
x
→
indet
layout
x
VAL,ALG
u

The function tov is defined as follows:

$$\text{tov} : \{\underline{\text{int}},\underline{\text{rat}},\underline{\text{alg}},\underline{\text{real}}\} \to \{\text{INT},\text{RAT},\text{ALG},\text{REAL}\}$$

$$\text{tov}(\underline{\text{int}}) = \text{INT}$$

$$\text{tov}(\underline{\text{rat}}) = \text{RAT}$$

$$\text{tov}(\underline{\text{alg}}) = \text{ALG}$$

$$\text{tov}(\underline{\text{real}}) = \text{REAL}.$$

9.2 Substitution evaluation

a) A substitution is a primary that is defined for algebraic variables in which we would like to substitute some algebraic values for the indeterminates of this algebraic or some of them. In general, it consists of several levels, i.e. we have a substitution in a substitution etc....
So in the general case we have to go to the lowest level of substitution to perform it first, then go up gradually (1,2).

When we are at this lowest level of substitution (1) a pairing of each indeterminate and its corresponding argument is performed using one of two macros, pair or pairing, that are described formally below. These two marcros make use of another macro, arglay (argument layout). What arglay does is, given a list of pairs (the first one is the form or indeterminate, the second the argument to replace the indeterminate):

(1) To find the eventual layout of the second element of each pair.

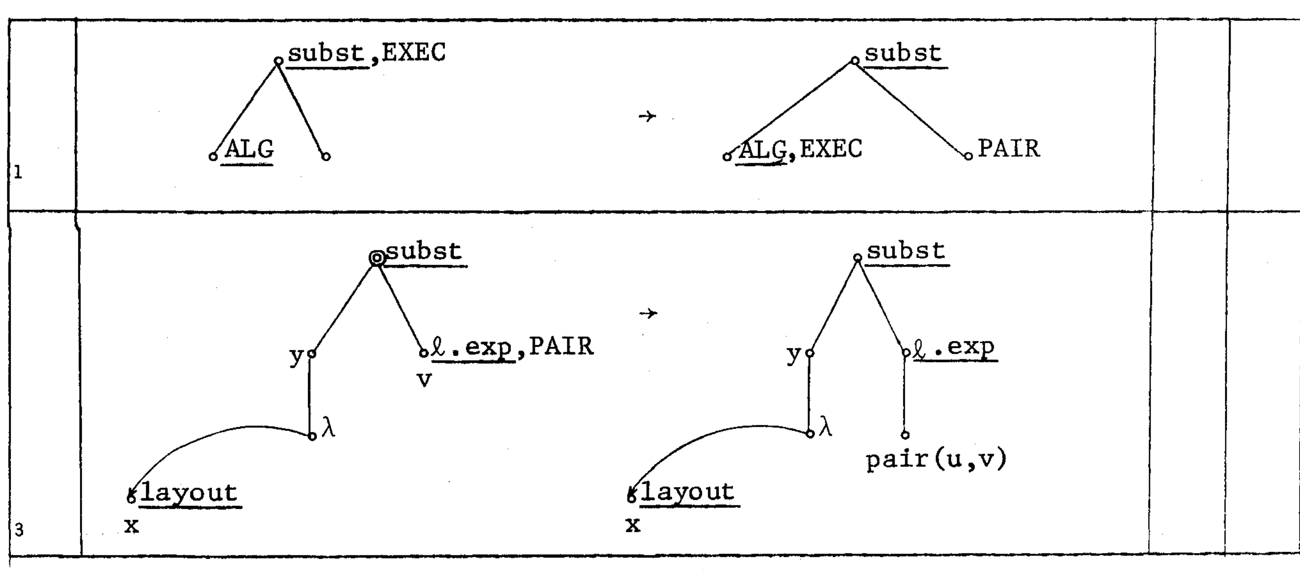
(2) To make sure that the layouts, whenever they exist, are the same for all the arguments.

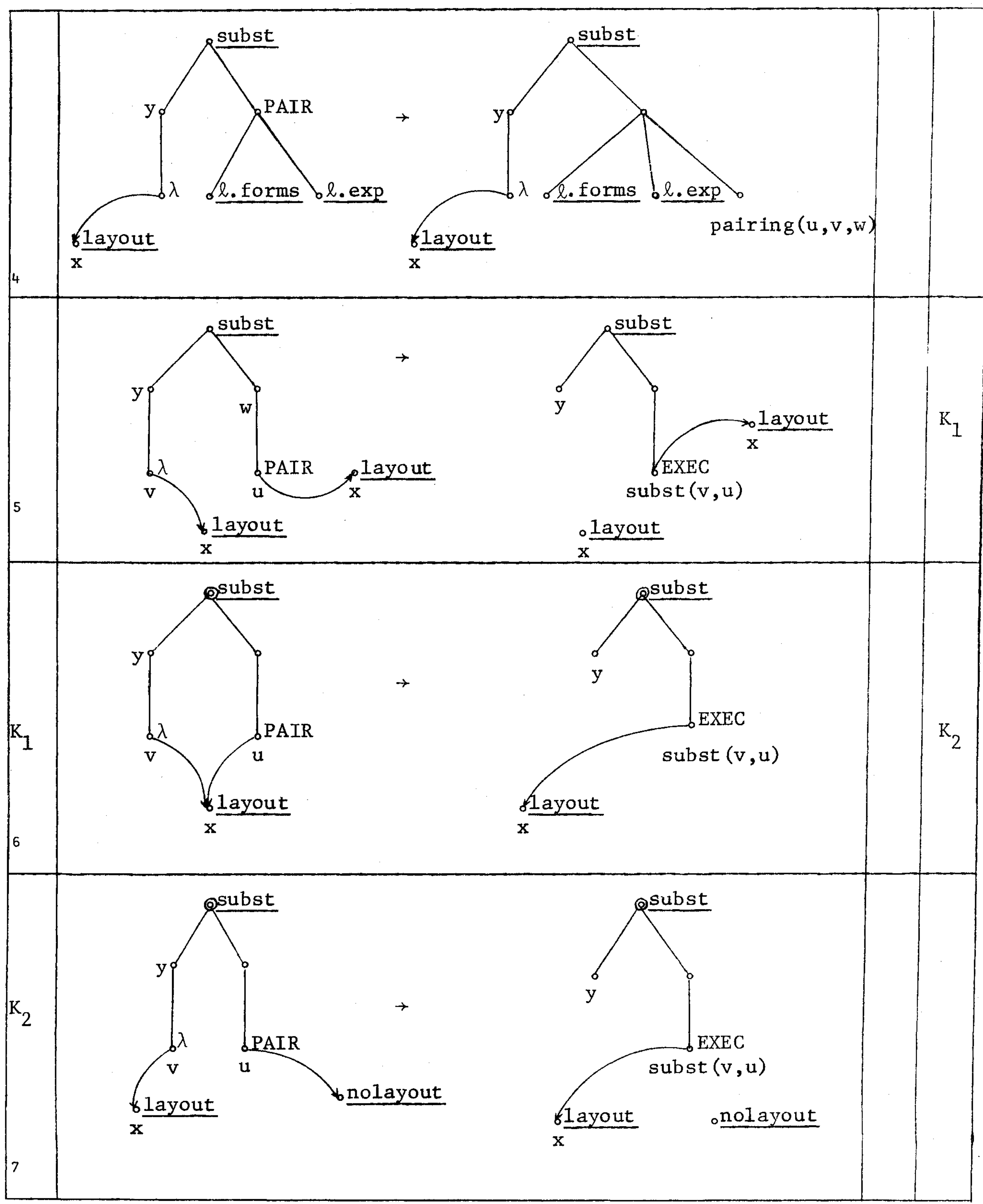
(3) To link the layout, common for all the arguments, to the top node of the list of pairs.

The common layout when it exists, is obtained after performing the macro arglay. Then it is used to decide about the layout of the result of the substitution (5-9). The layout of the substitution is the common layout to the arguments and algebraic variable if it is the case. Otherwise, if there is no layout for the arguments (e.g. numeric arguments) then the common layout will be that of the variable. Otherwise it will be that of the arguments on the condition that all indeterminates that appear in the layout of the variable be substituted (8,9).

The substitution itself is given as a macro, subst, described formally below. What it does is to replace each indeterminate by the corresponding argument given in the pair (indeterminate, argument).

b)





4
subst
y
PAIR
λ
ℓ.forms
ℓ.exp
layout
x
→
subst
y
λ
ℓ.forms
ℓ.exp
layout
x
pairing(u,v,w)
5
subst
y
w
λ
v
PAIR
u
layout
x
layout
x
→
subst
y
layout
x
EXEC
subst(v,u)
layout
x
K1
K1
6
subst
y
λ
v
PAIR
u
layout
x
→
subst
y
EXEC
subst(v,u)
layout
x
K2
K2
7
subst
y
λ
v
PAIR
u
layout
x
nolayout
→
subst
y
EXEC
subst(v,u)
layout
x
nolayout

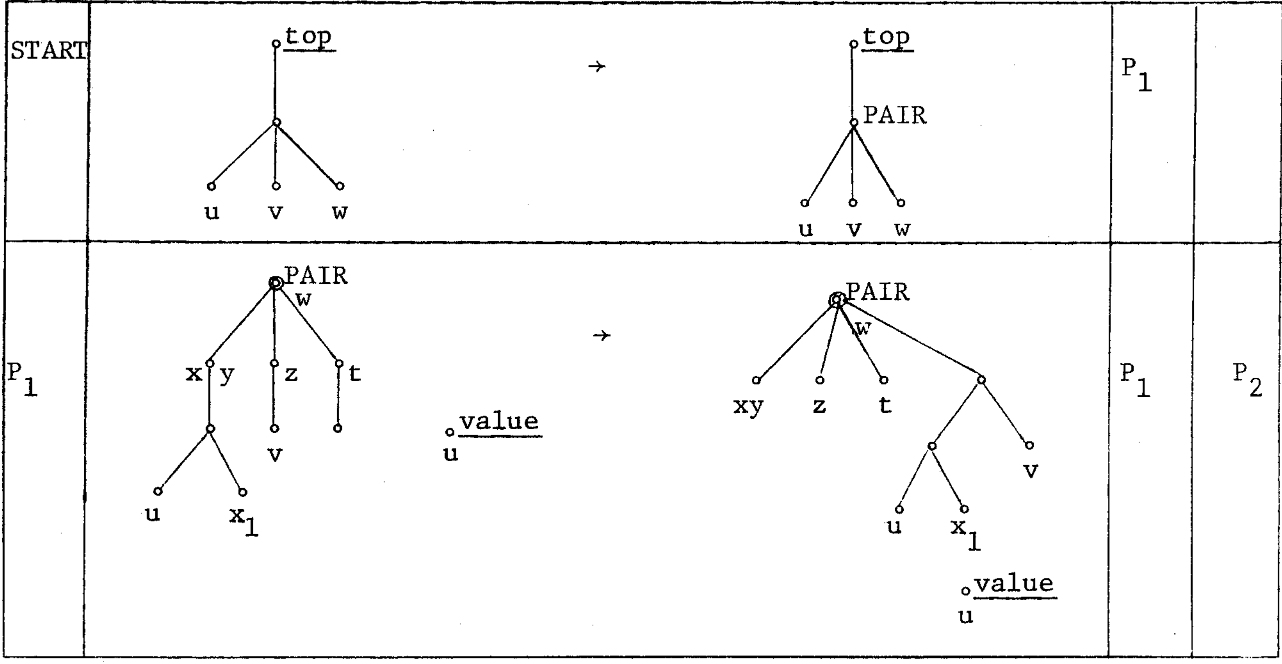
K_3 8	PAIR ; blank → layout x	ERROR	K_4
9	subst ; y w ; λ v ; layout x ; PAIR u ; layout z → subst ; y ; layout x ; EXEC subst (v,u) ; layout z		
10	subst ; y w ; u ; v ; layout x → subst ; y w v λ ; layout x		

The macro pair is defined by:

START	top ; u v → top ; PAIR ; u v	P_1	
P_1	PAIR ; w ; u v ; x y → PAIR ; w ; u v ; x y	P_1	P_2

P_2	PAIR w v x	→		ERROR	P_3
P_3	PAIR w u x	→	PAIR w u x blank	P_3	P_4
P_4	top PAIR w	→	arglay(w)	STOP	ERROR

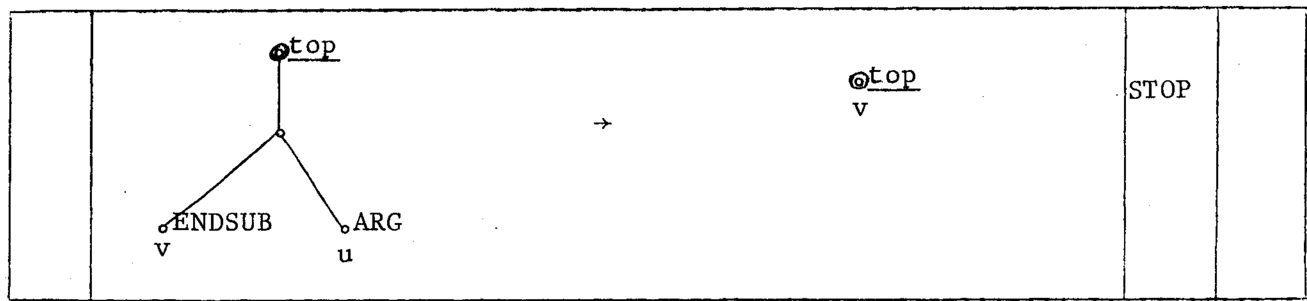
The macro operation pairing is defined by:



P_1	PAIR w x y z t v u u x_1	→	PAIR w xy z t v u x_1	P_1	P_2
P_2	PAIR w u v x	→		ERROR	P.
P_3	PAIR w u v x	→		ERROR	P_4
P_4	PAIR w v u x y	→		ERROR	P_5
P_5	PAIR w u x	→	PAIR w u u <u>blank</u>	P_5	P_6
P_6	<u>top</u> PAIR x	→	<u>top</u> arglay(x)	STOP	ERROR

Finally the macro subst is defined by:

START	top v u	→	top SUBST v ARG u		
	ν, SUBST	→	ν SUBST		
	ν ENDSUB	→	ν SUBST		
	ν ENDSUB	→	ν, ENDSUB		
	$\ominus$, SUBST	→	$\ominus$ SUBST		
	$\ominus$ ENDSUB	→	$\ominus$, ENDSUB		
	$\circledcirc$ indet, SUBST u ARG v u	→	$\circledcirc$ indet, ENDSUB v ARG v u		



9.3 Evaluation of addition, subtraction, multiplication and division

a) First if one of the two operands in any arithmetic operation is short and the other long then convert the short operand to a long one (1,2).

We can evaluate one of the four elementary arithmetic operations only if the two operands (values) are of the same type.

For the case of algebraic operands, the layout of each of them should be identical to that of the other before the operation can be performed (6,7).

For integers and reals the following basic functions are used and their formal definition is given below:

- irop (integer arithmetic operations): This function has three arguments, an arithmetic operator and two integers. Its value is the result of applying this operator to the given integers.
- ineg (integer negation): This function gives the negation of an integer.
- rarop (real arithmetic operations): This function has three arguments, an arithmetic operator and two real numbers. Its value is the result of applying the operator to the given reals.
- rneg (real negation): This function gives the negation of its real argument.

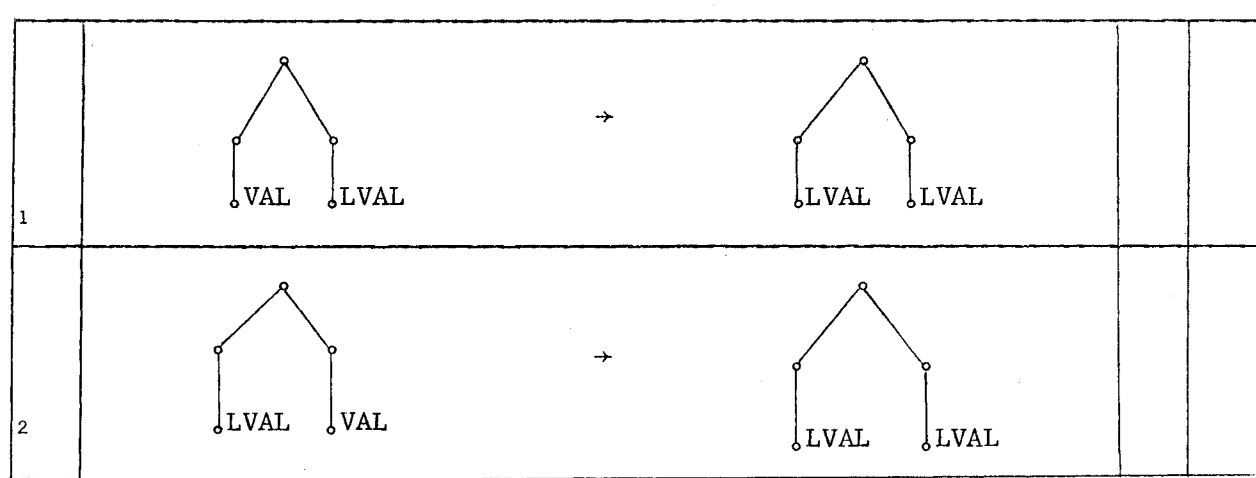
For algebraics and rationals some macro-operations are used. They are the following:

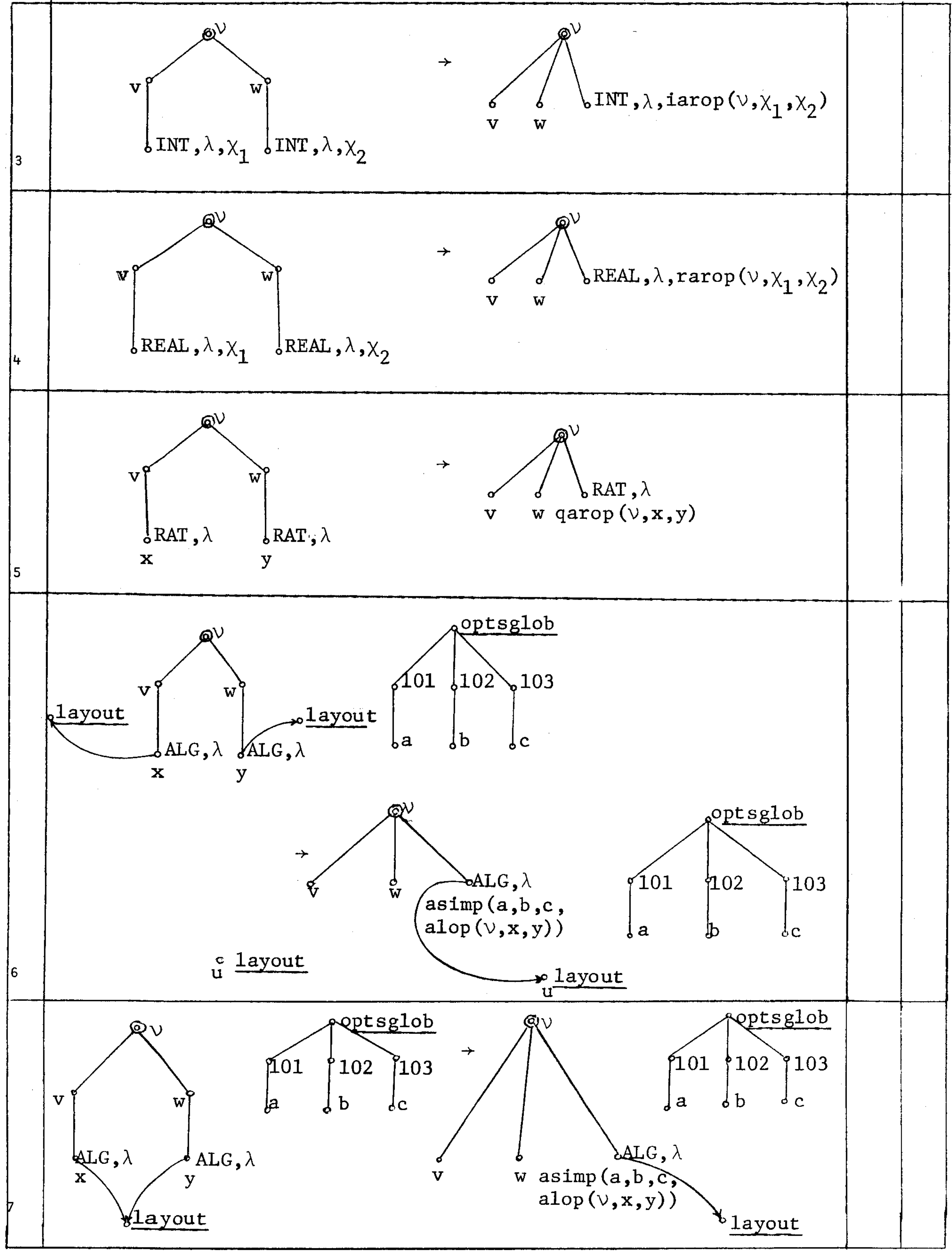
qarop: Given an arithmetic operation (+,-,*,/) and two representations of rational numbers, this macro operation gives a tree which is a representation of a rational fraction result of the arithmetic operation on these rationals.

alop : Given an operation in the set {+,-,*,/} and two trees each representing an algebraic, this macro operation gives a tree representing the algebraic resulting from the two given algebraics by application of this operation.

asimp: Given the algebraic options and a tree representing an algebraic, this macro operation transforms the tree into a simplified one representing the same value of the algebraic. The degree of simplification depends on the specified options. (For a complete description of these options and their meaning see [1], D3 and L7).

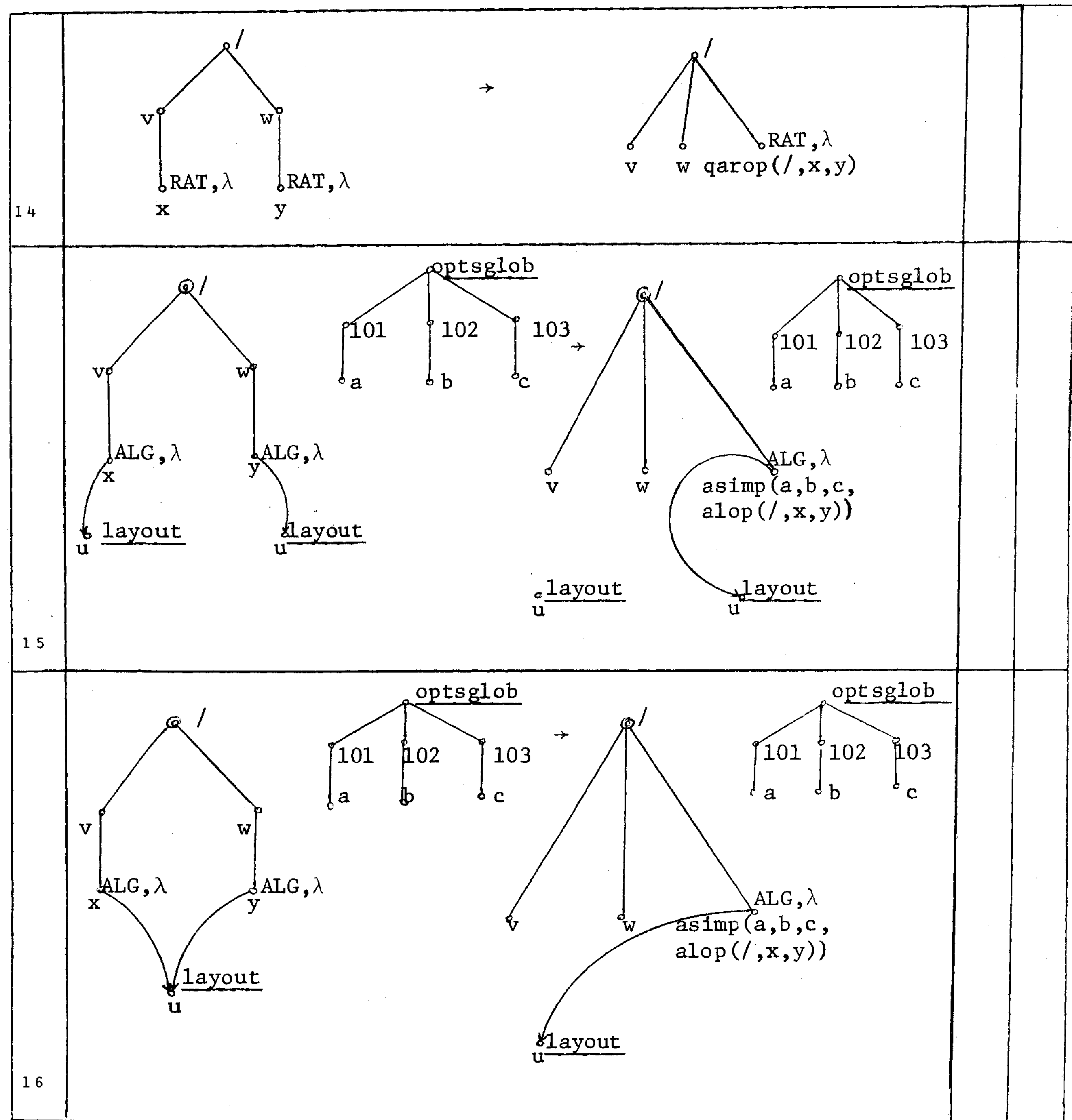
b)





3
v
w
INT,λ,χ1
INT,λ,χ2
→
INT,λ,iarop(ν,χ1,χ2)
4
REAL,λ,χ1
REAL,λ,χ2
REAL,λ,rarop(ν,χ1,χ2)
5
RAT,λ
x
y
w qarop(ν,x,y)
6
layout
ALG,λ
optsglob
101
102
103
a
b
c
asimp(a,b,c,
alop(ν,x,y))
u layout
7

No.	Pattern		Replacement		
8	$\ominus$; v ; INT, λ, χ	→	$\ominus$; v ; INT, λ, ineg(χ)		
9	$\ominus$; v ; REAL, λ, χ	→	$\ominus$; v ; REAL, rneg(χ)		
10	$\ominus$; /, RAT, λ ; χ_1 ; χ_2	→	$\ominus$; v ; /, RAT, λ ; ineg(χ_1) ; χ_2		
11	$\ominus$; ALG, λ ; v ; layout	→	$\ominus$; $\ominus$, ALG, λ ; u ; layout		
12	/ ; v ; w ; INT, λ, χ_1 ; INT, λ, χ_2	→	/ ; v ; w ; /, RAT, λ ; χ_1 ; χ_2		
13	/ ; v ; w ; REAL, λ, χ_1 ; REAL, λ, χ_2	→	/ ; v ; w ; REAL, λ ; rarop(/, χ_1, χ_2)		


14
/
v
w
RAT,λ
RAT,λ
x
y
→
/
RAT,λ
v
w
qarop(/,x,y)
15
/
v
w
ALG,λ
ALG,λ
x
y
layout
layout
u
u
optsglob
101
102
103
a
b
c
→
/
v
w
ALG,λ
asimp(a,b,c,
alop(/,x,y))
layout
layout
u
u
optsglob
101
102
103
a
b
c
16
/
v
w
ALG,λ
ALG,λ
x
y
layout
u
optsglob
101
102
103
a
b
c
→
/
v
w
ALG,λ
asimp(a,b,c,
alop(/,x,y))
layout
u
optsglob
101
102
103
a
b
c

The basic functions used here are:

$$\text{ineg} : Z \to Z$$
$$z \to -z$$

$$\text{rneg} : R \to R$$
$$r \to -r$$

$$\text{iarop} : \{+,-,*,/\} \times Z \times Z \to Z$$
$$(\nu, z_1, z_2) \to z_1 \; \nu \; z_2$$

$$\text{rarop} : \{+,-,*,/\} \times R \times R \to R$$
$$(\nu, r_1, r_2) \to r_1 \; \nu \; r_2$$

9.4 Evaluation of Exponentiation

a) For an integer exponent the result should be of the same type as the base (1-4).

An algebraic can only have an integer exponent (3).

If the exponent is not integer the result is real. So, a transformation to real of the base and exponent is done if they are not already real. Then the exponentiation is performed (5-10).

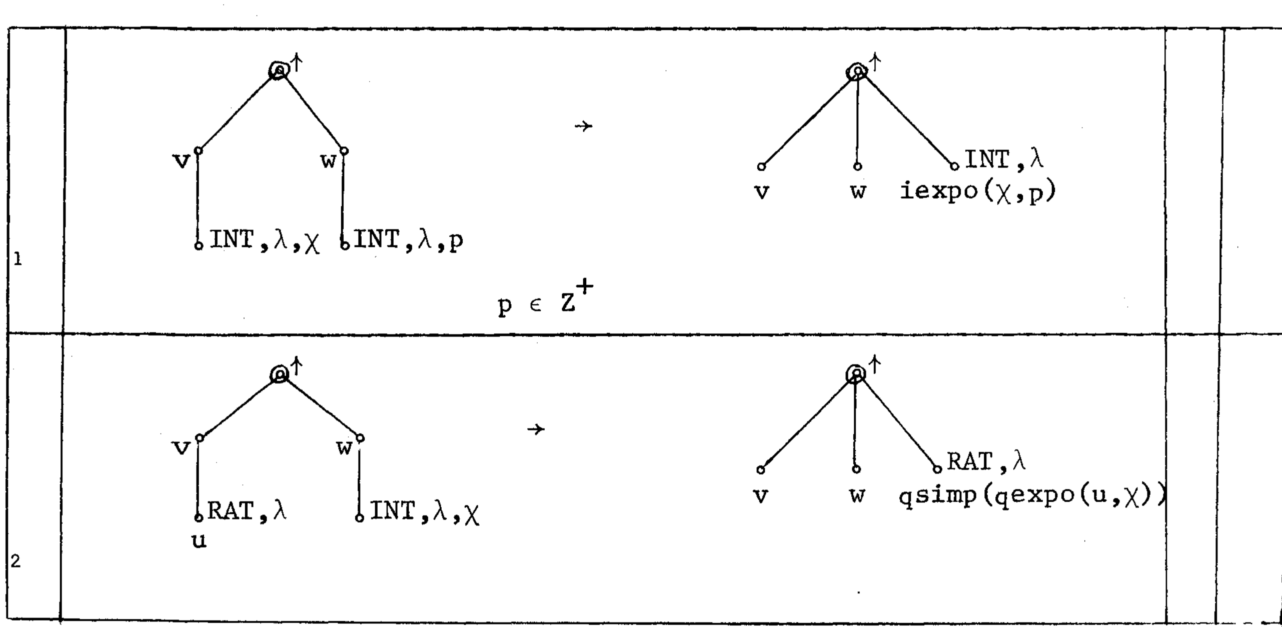
The functions that are used here are:

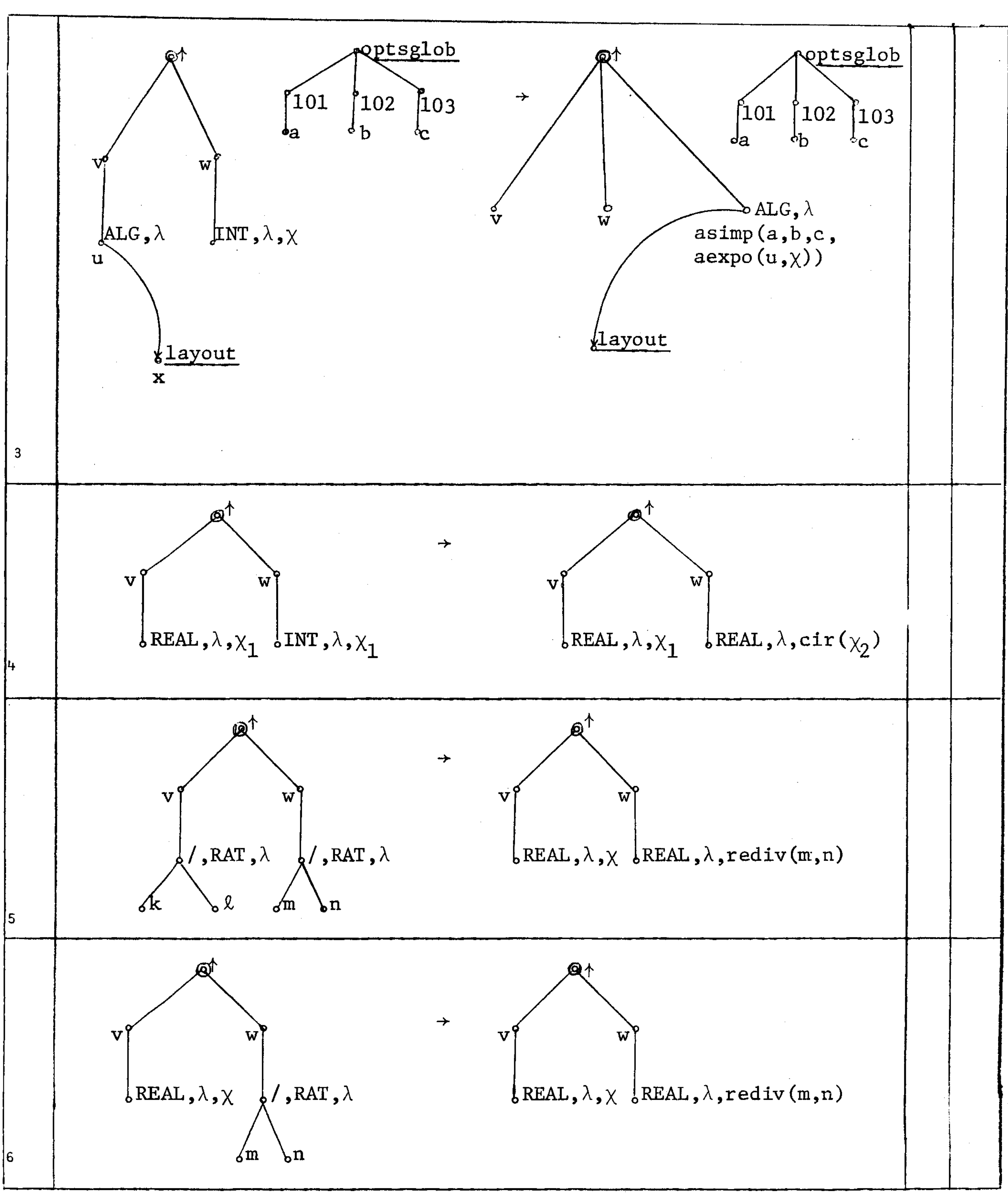
- rediv (for real division) previously defined.
- cir (for conversion from integer to real) also previously defined.
- iexpo (for integer exponentiation) has two integer arguments the second one being positive. It computes the exponentiation as defined formally below.
- rexpo (for real exponentiation) has two real arguments and it computes the exponentiation as defined formally below.

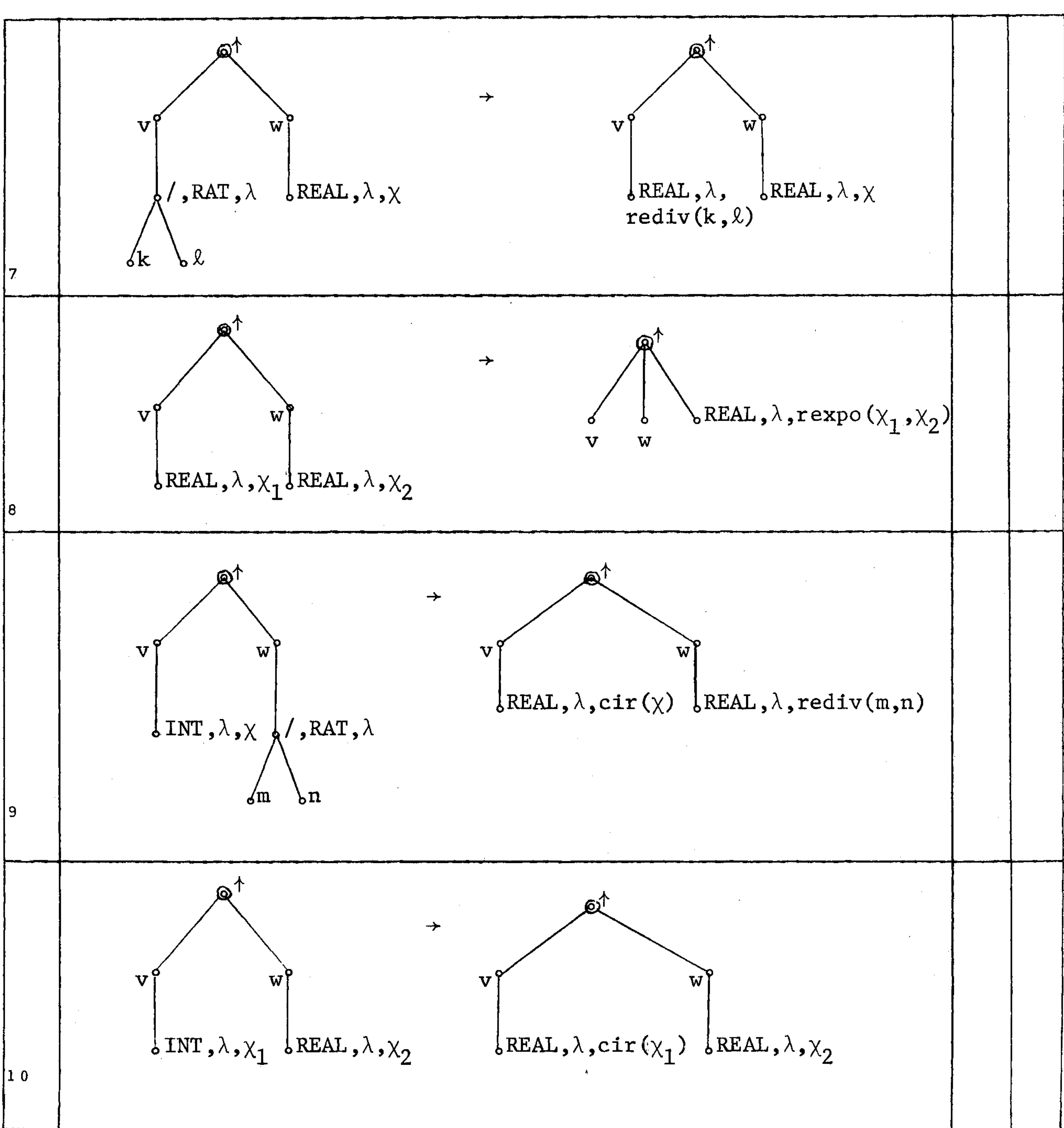
The new macros used here are:

- qexpo (for rational exponentiation): Given a tree representing a rational fraction and an integer this macro yields a tree representing the rational fraction resulting from exponentiation of the numerator and denominator of the given rational fraction.
- aexpo (for algebraic exponentiation): Given a tree representing an algebraic and an integer this macro gives a tree representing the algebraic that results from exponentiation.
- qsimp: Macro for simplification of rational numbers. Given a tree representing a rational, this macro yields a tree representing the simplified rational number equivalent to the given one.

b)




3
optsglob
101
102
103
a
b
c
v
w
ALG,λ
u
INT,λ,χ
layout
x
→
optsglob
101
102
103
a
b
c
v
w
ALG,λ
asimp(a,b,c,
aexpo(u,χ))
layout
4
v
w
REAL,λ,χ1
INT,λ,χ1
→
v
w
REAL,λ,χ1
REAL,λ,cir(χ2)
5
v
w
/,RAT,λ
/,RAT,λ
k
ℓ
m
n
→
v
w
REAL,λ,χ
REAL,λ,rediv(m,n)
6
v
w
REAL,λ,χ
/,RAT,λ
m
n
→
v
w
REAL,λ,χ
REAL,λ,rediv(m,n)



7
v
w
/,RAT,λ
REAL,λ,χ
k
ℓ
→
v
w
REAL,λ,
rediv(k,ℓ)
REAL,λ,χ
8
v
w
REAL,λ,χ1
REAL,λ,χ2
→
v
w
REAL,λ,rexpo(χ1,χ2)
9
v
w
INT,λ,χ
/,RAT,λ
m
n
→
v
w
REAL,λ,cir(χ)
REAL,λ,rediv(m,n)
10
v
w
INT,λ,χ1
REAL,λ,χ2
→
v
w
REAL,λ,cir(χ1)
REAL,λ,χ2

The following new functions are defined:

$$\text{iexpo} \quad : Z \times Z^{+} \to Z$$

$$(z_1, z_2) \to z_1^{z_2}$$

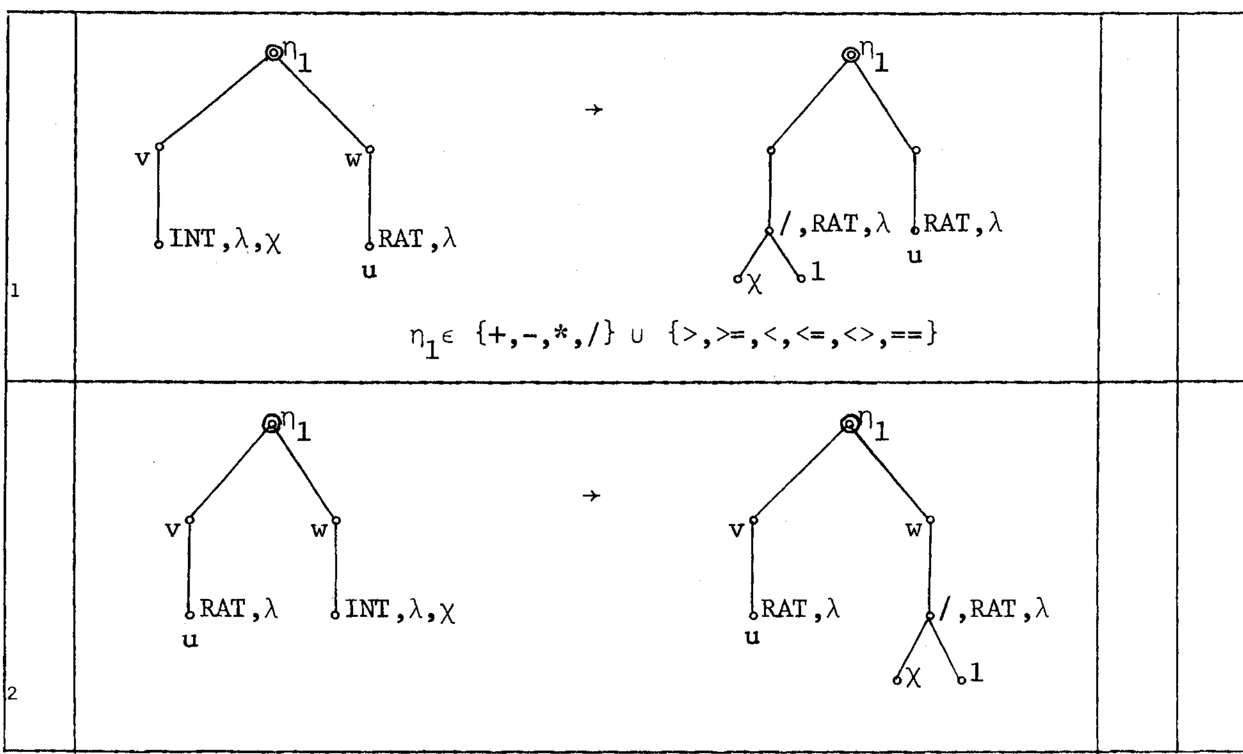
$$\text{rexpo} \quad : R^{+} \times R \to R$$

$$(r_1, r_2) \to r_1^{r_2}$$

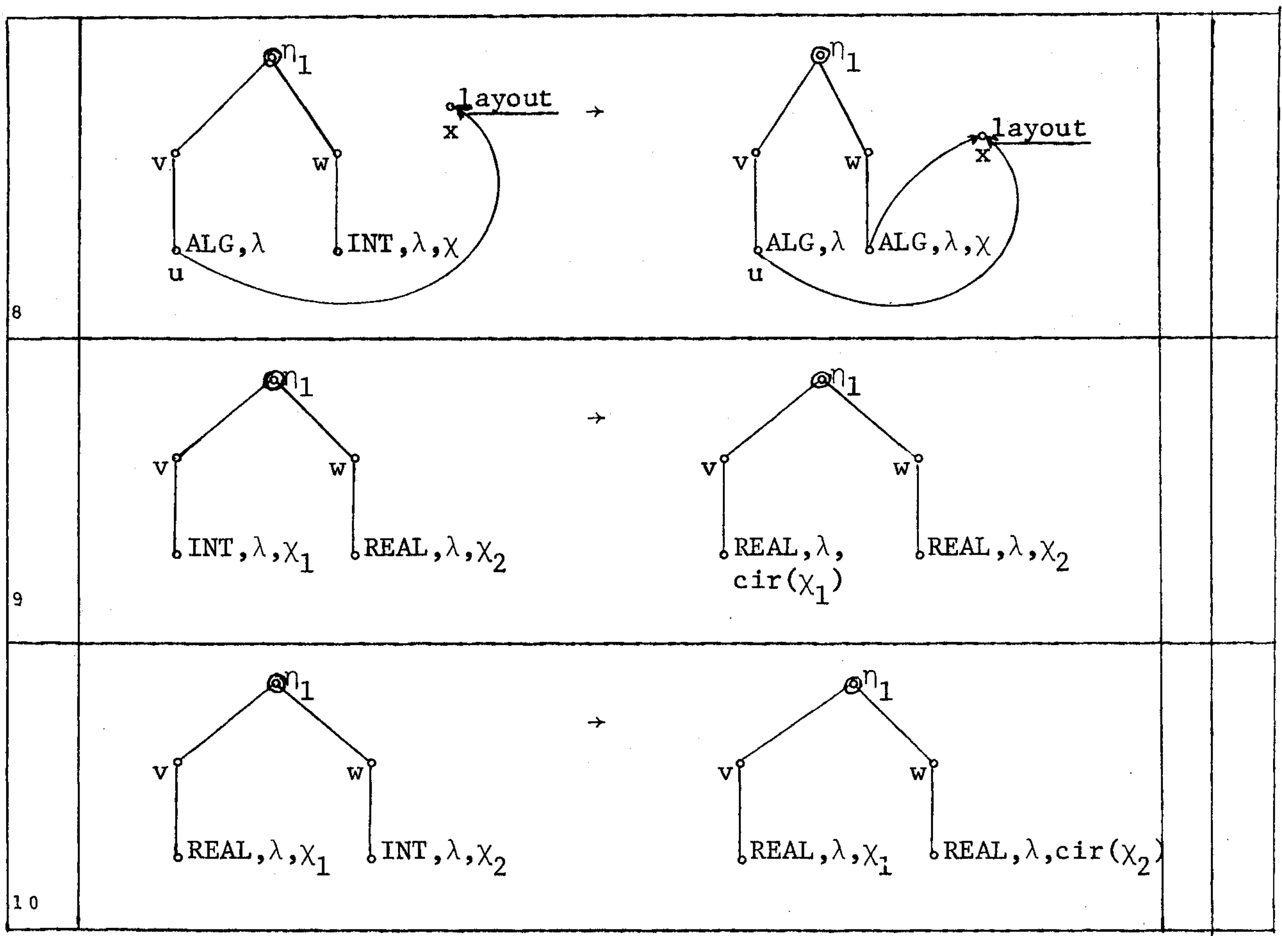
9.5 Conversion of types in expressions

a) When we have either an arithmetic operation or comparison between two operands of different types it is possible to convert some types to others. More precisely if the types are comparable the value of lower type in the hierarchy is always converted to the higher type.

b)



No.	Left side		Right side		
3	η_1; v, w; x: RAT, λ; y: ALG, λ; u: layout	→	η_1; v, w; x: ALG, λ; y: ALG, λ; u: layout		
4	η_1; v, w; x: ALG, λ; y: RAT, λ; u: layout	→	η_1; v, w; x: ALG, λ; y: ALG, λ; u: layout		
5	η_1; v, w; /, RAT, λ; REAL, λ, χ; k, ℓ	→	η_1; v, w; REAL, λ, rediv(k,ℓ); REAL, λ, χ		
6	η_1; v, w; REAL, λ, χ; /, RAT, λ; k, ℓ	→	η_1; v, w; REAL, λ, χ; REAL, λ, rediv(k,ℓ)		
7	η_1; v, w; INT, λ, χ; u: ALG, λ; x: layout	→	η_1; v, w; ALG, λ, χ; u: ALG, λ; x: layout		



9.6 Overflow

a) When a function is not defined for the given arguments the value is outofdomain. In this case an overflow or a loverflow occurs depending on the precision of the calculations. The execution terminates with an error.

b)

1	VAL, out-of-domain	→	overflow	ERROR	
2	LVAL, out-of-domain	→	loverflow	ERROR	

10. Relation - Comparison

a) After we obtain the same type for the operands of a relation or comparison (9.4) we can effectively compare the operands.

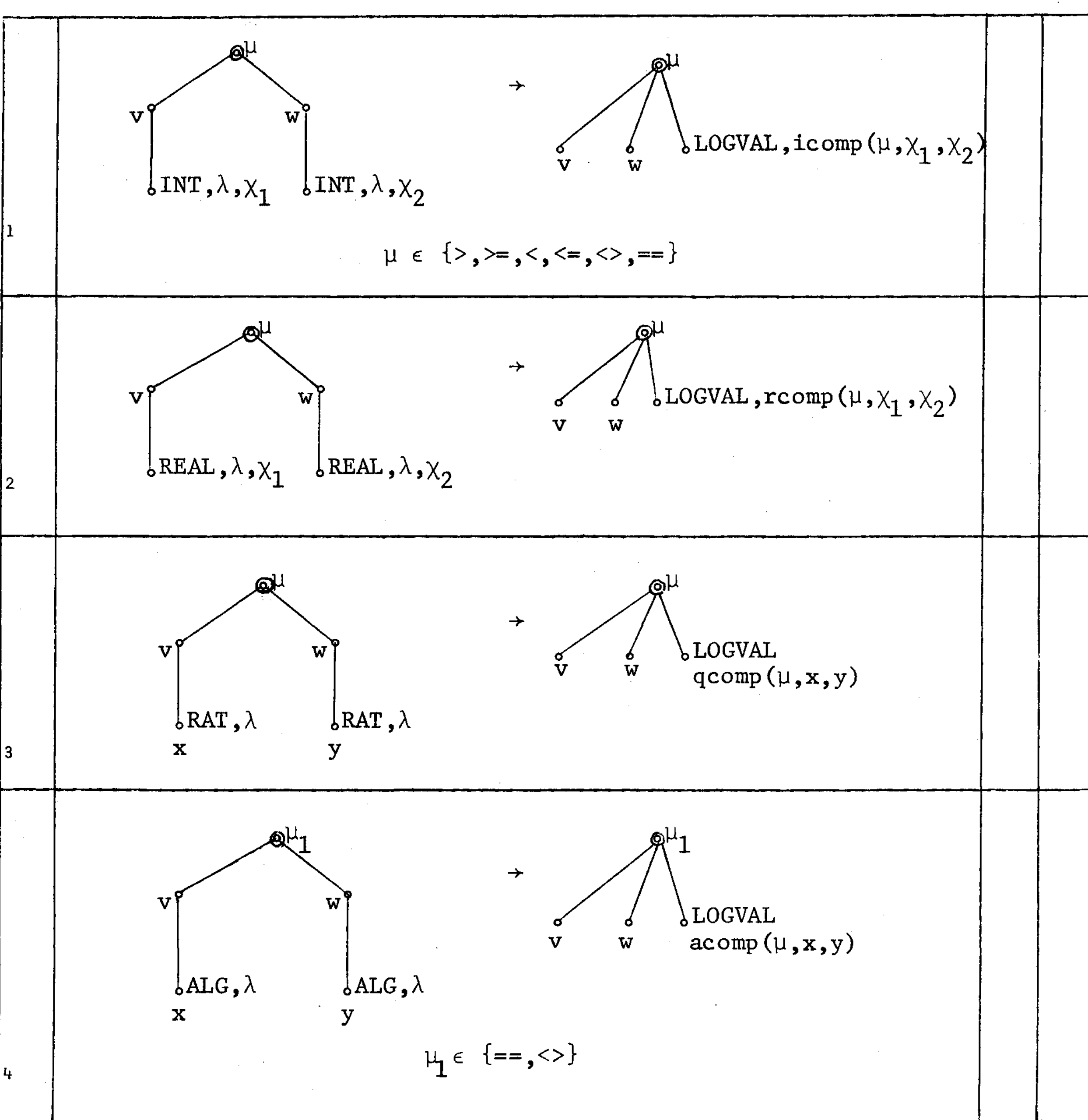
For integer and real operands, functions are used to perform the comparison, their result being logical. These functions are:

- icomp (for integer comparisons) that has three arguments the first one is a relation symbol and the other two integers to be compared. The result is _true_ if the relation holds between these integers, _false_ otherwise.
- rcomp (for real comparisons) as for icomp except that it is for real operands instead of integer operands.

For rational and algebraic arguments of a relation macro operations are used.

- qcomp: Macro for rational comparison; it compares two rational numbers given by their tree representation. The comparison symbol constitutes the first argument for this macro,the other two being the numbers. The result is _true_ if the relation holds, _false_ otherwise.
- acomp: Macro for algebraic comparisons; as for qcomp except that the trees represent algebraics instead of rationals.

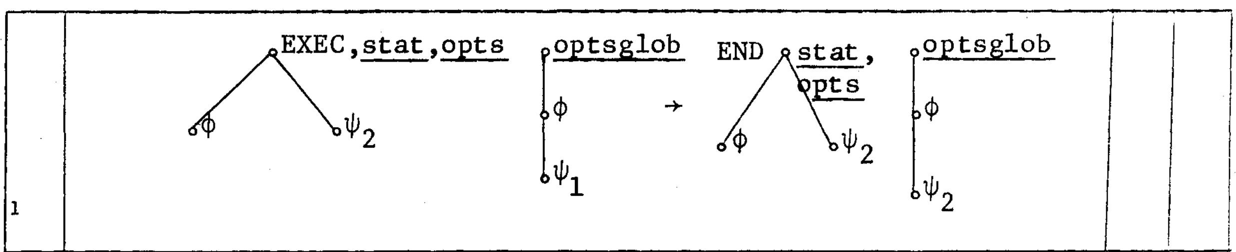
b)



11. Algebraic options statement

a) When executed, the option statement should set a label of the global options tree to the wanted value. So that whenever we want in the program, we can change the options, thus changing the degree of simplification of algebraics from this point on.

b)



12. Procedure invocation

12.1 The invocation statement

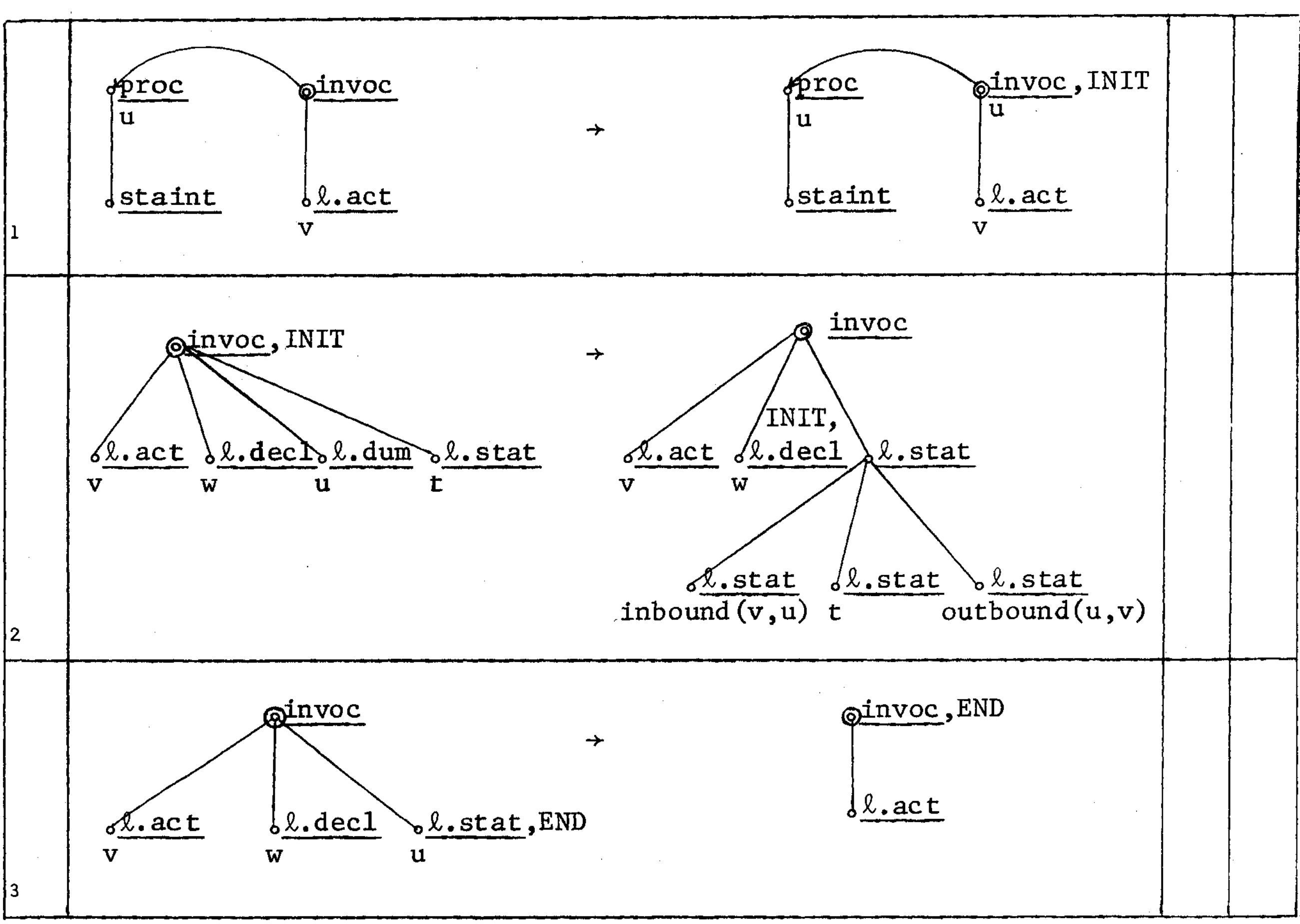
a) When a procedure is invoked the whole of the procedure tree except the subtree corresponding to the static internal variables (labeled staint) is copied into the invocation tree (1).

Afterwards, the trees for the inbound and outbound transmission of arguments are constructed using two macro operations (described in the following paragraphs), that are inbound and outbound.

The initialization of the declared variables of this procedure follows by putting the keyword INIT on the top node (2).

When the execution of the procedure ends and if it is an invocation statement (not a function primary) then all what is not in the list of actual parameters will be deleted to come back to the original form of the invocation statement tree (3). For the function this is done after computing the value of the function (4.b, 11-14).

b)



12.2 Macro Inbound

a) Given a list of dummies and a list of actuals this macro assigns to each dummy the corresponding actual. If there is no corresponding actual it puts blank. If there is no dummy variables but there is some actual variables or if the number of actuals is greater than that of the dummies an error is detected. Its formal description is given below.

b)

START	top ℓ.dum ℓ.act v blank	→	top ℓ.dum ℓ.act v	L_3	L_4
L_1	w top ℓ.dum ℓ.act u v x y	→	w top =,stat ℓ.dum ℓ.act x y u v	L_1	L_2
L_2	w top ℓ.dum ℓ.act	→	w	STOP	L_3
L_3	w top ℓ.dum ℓ.act	→		ERROR	L_4
L_4	w top ℓ.dum ℓ.act	→	w	STOP	

12.3 Macro Outbound

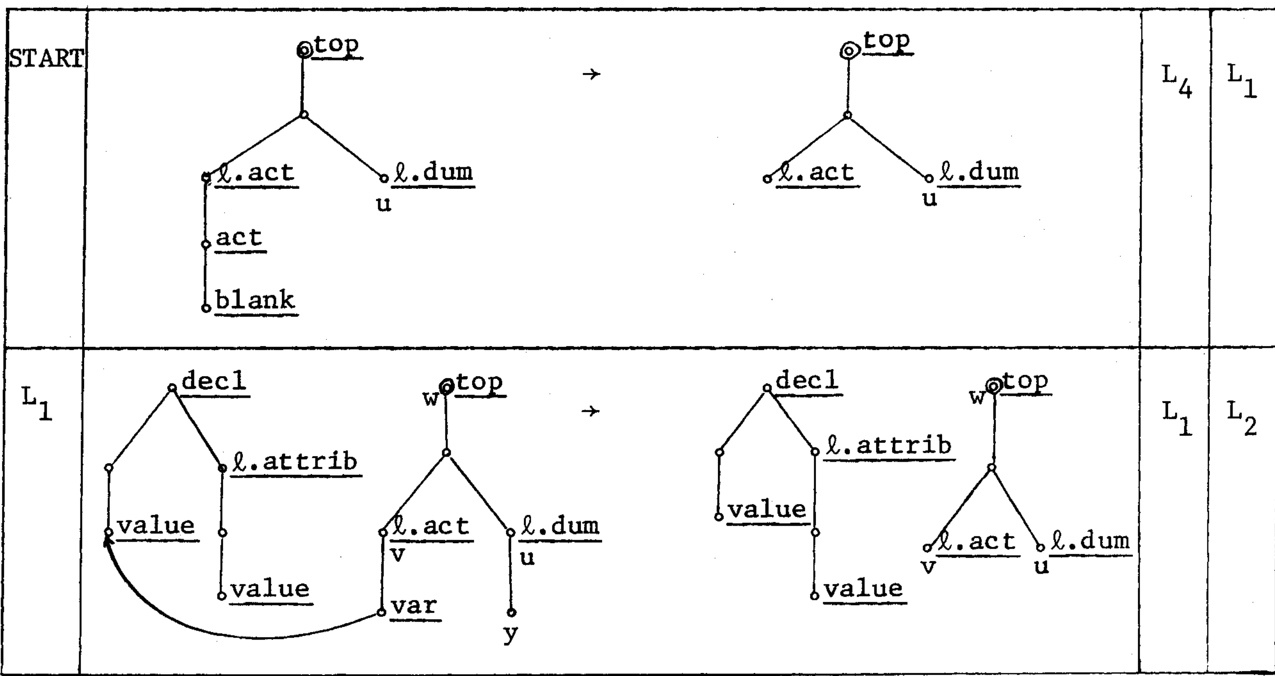
a) Given a list of actuals and a list of dummies this macro assigns the value of each dummy into the corresponding actual if it is a variable and the dummy does not have the value attribute (dummy labels have the value attribute).

If the list of actuals is empty then there is no outbound transmission to perform.

If the number of actuals is less than that of the dummies we just stop after the last actual.

The number of actuals cannot be greater than that of the dummies as it was checked in the inbound transmission before. So there is no need to check it again.

b)

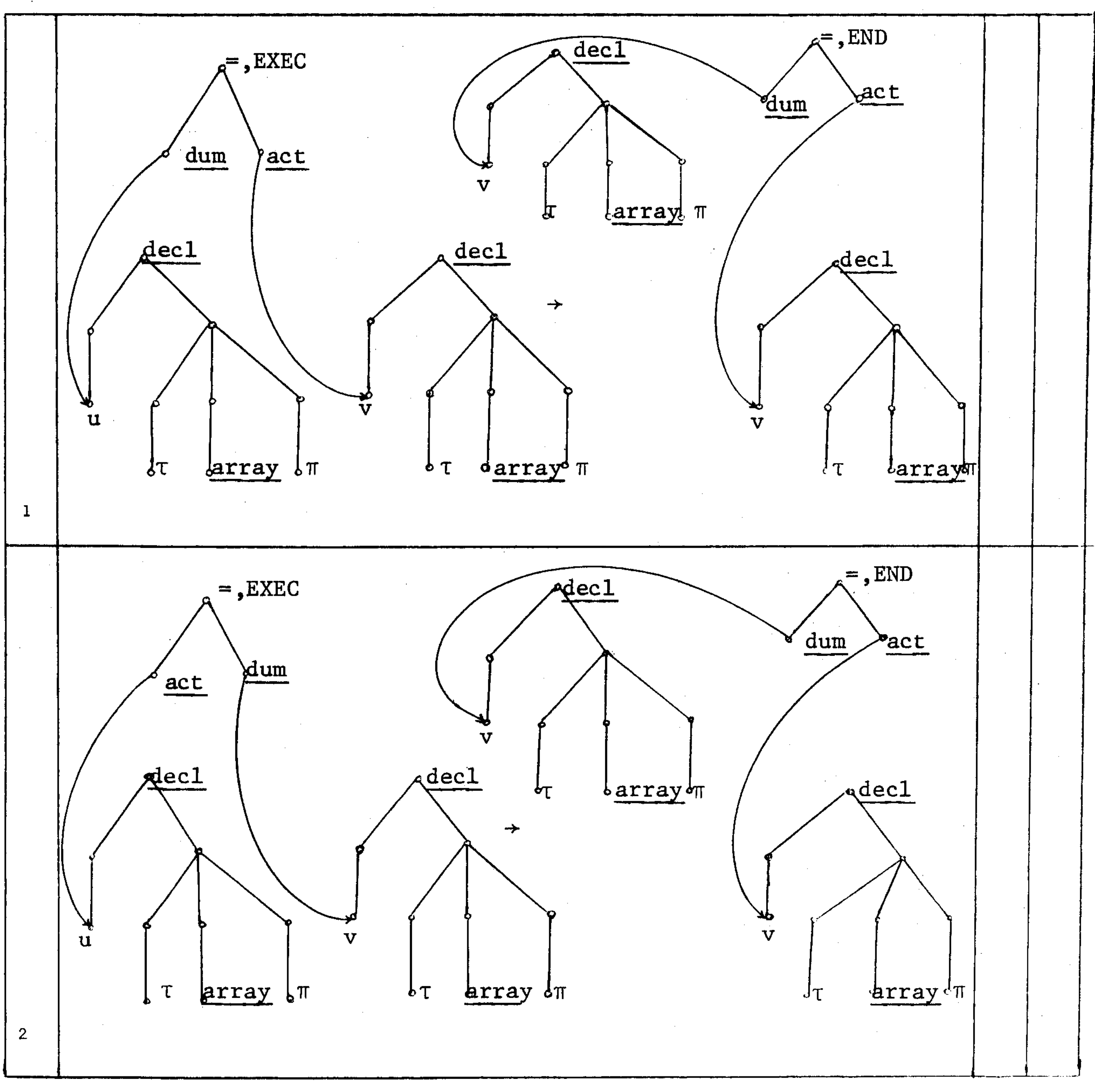


L_1	w top; decl; ℓ.attrib; value; ℓ.act v; elem x; ℓ.dum u; y	→	w top; decl; ℓ.attrib; value; ℓ.act v; ℓ.dum u	L_1	L_2
L_2	w top; ℓ.act v; ℓ.dum u; x; y	→	w top; =,stat; x; y; ℓ.act v; ℓ.dum u	L_1	L_3
L_3	w top; ℓ.act v; ℓ.dum u; x; y	→	w top; ℓ.act v; ℓ.dum u	L_1	L_4
L_4	w top; ℓ.act v; ℓ.dum u	→	w	STOP	L_5
L_5	w top; ℓ.act; ℓ.dum	→	w	STOP	

12.4 Arrays in inbound and outbound transmission

a) Arrays as a whole can be transmitted as actual or dummy variables and we must perform consequently an assignment in one direction or the other. But they must have the same type, precision and identical descriptor blocks to make the assignment possible.

b)



13. The Input/Output statements

a) When a read statement is to be executed, the first element of the input data is evaluated. When the value of this data expression is found, its value is assigned to the variable.

When we want to write the value of a variable this value is transferred to the output list.

If we want to read and there is no more input data then the execution stops.

b)

1		read,EXEC u ∘	→		read u READ		
2	READ	input w datastat	→	READ	input w EXEC		
3	read u READ,var x	input ∘ λ,τ v	→	read u =,stat var x λ,τ v	input DEL λ,τ v		
4	read u READ,elem x	input ∘ λ,τ v	→	read u =,stat elem,SELECT x λ,τ v	input DEL λ,τ v		

5	input, u, DEL, x, λ, y → input, u			
6	read, u, =,stat,END, x, y → read, u, REND, x			
7	read, u, v, REND → read, u, v, READ			
8	u, read, REND → u, read,END			
9	READ input → READ input		STOP	
10	write,EXEC, u, arg → write, u, arg,EXEC			
11	write, string,EXEC, u, output, w → write, string,WEND, w, output, u			
12	write, u, arg, λ_1, v, output, w → write, u, DEL, λ_1, v, w, output, v			

13	write (node); w̄; w: DEL; v: λ_1 → write (node); w; u: arg,WEND			
14	write (node) with children WEND, (node) → write (node) with children (node), EXEC			
15	u: write; WEND → u: write,END; (node)			

REFERENCES

1. W.S. Brown, ALTRAN users' manual, Bell Telephone Laboratories, New Jersey, 07974.

2. K. Culik II, A Model for Formal Definition of Programming Languages, Department of Applied Analysis and Computer Science, University of Waterloo, Research Report CSRR 2065, June 1972.

3. A.D. Hall et al., ALTRAN System for Rational Function Manipulation - a survey, CACM 14, 8(1971).

4. D.J. Rosenkrantz, Programmed Grammars and Classes of Formal Languages, JACM, 16, 107-131 (1969).

INDEX

	Page
<actual>	51
acomp	146
aexpo	140
ALG	73
alop	135
arglay	127
<array var>	51
asimp	135
<attrib>	33
<attrib stat>	35
<back ref>	86
<body>	56
<char string>	49
check	117
cir	113
constr	94
<control var>	53
<D.B.>	32
<data exp>	85
<data stat>	85

	Page
<digit>	24
<doend-stat>	49
<do-group>	54
<do-stat>	53
<dum>	51
<dummy label>	48
END	90
ENDIN	92
<end-stat>	49
EXEC	88
<exp>	42
<form>	39
<group>	55
<heading>	59
iarop	139
icomp	146
<ident>	24
iexpo	143
<if-group>	55
inbound	149

Page

<indet> 32

ineg 139

INIT 90

<input> 86

INT 73

<int const> 26

<int exp> 43

intsh 113

<invocation> 51

<I/O arg> 49

<job> 88

<label> 46

<label var> 46

<label exp> 47

<ℓ.attrib stat> 37

<layout> 32

<letter> 24

<ℓ.exp> 42

<log const> 30

<log exp> 45

LOGVAL 96

<log var> 44

	Page
LPRIM	101
LVAL	98
<ℓ.var>	50
outbound	151
<output>	86
pair	130
pairing	131
<parm>	53
<precision>	33
PRIM	101
<proc>	67
<prog>	81
qarop	135
qcomp	146
qexpo	140
qsimp	140
rarop	139
rcomp	46
RAT	73
REAL	73
<real const>	27
realsh	113
rediv	113

	Page
<relation>	44
rexpo	143
rneg	139
<scope>	33
SELECT	101
SELEND	103
<s int const>	27
<subject>	34
subst	133
<subst>	40
<stat>	48
<st class>	33
STOP	90
<struct>	33
tov	127
<type>	32
<unlabeled group>	55
VAL	98
<var>	39