

Linked Forest Manipulation Systems
a Tool for Computational Semantics[†]

K. Culik II
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

M. Farah
Department of Physics and Mathematics
University of Moncton
Moncton, N. B., Canada

December 1977

Research Report CS-77-18

[†] This research was supported by the National Research Council of Canada
Grant No. A7403.

Abstract

Linked forest manipulation systems were introduced in [2] together with a definitional model for syntax and semantics of programming languages. Here we describe linked forest manipulating systems informally, and also give a new algebraically oriented formal definition. We then discuss their application to computational semantics.

Linked trees (forests) are capable to explicitly show all the syntactical properties of a programming language including such non-context-free properties as the correspondence between declarations and usages in a program. For this reason they seem to be the best data structure for computational semantics. We try to demonstrate that using forest manipulation systems we can get a relatively concise and at the same time readable formal definitions of real programming languages.

1. Introduction

Linked trees, as data structures for describing the computational semantics of programming languages, were introduced in [2], and seem to be best suited for that task. The definitional model introduced in [2] has been used to describe concisely and in readable form several complete programming languages (ALGOL 60 [13], ALTRAN [3], and LUCID [4]. In [4] it is also used to prove the correctness of an interpreter for LUCID.) Here, we give a readable informal description of the rewriting systems on linked trees, and a new somewhat-simplified algebraically oriented formal definition of linked forest manipulation systems and a description of their application to computational semantics. An ALGOL-like language ALG and λ -calculus are used as examples.

Computational semantics is the main motivation for studying these rewriting systems. There are various aspects which are desirable in a definitional model for the semantics of programming languages (e.g. rigorousness, readability, and conciseness). Linked tree (forest) manipulation systems ($\ell.f.m.s.$) give a tool which satisfies reasonably well all the above-mentioned requirements. Compared, for example, to the well-known Vienna definition method [7, 11], the model based on $\ell.f.m.s.$ allows an improvement in all these aspects. We will try to explain why this is so. But first, let us recall what is meant by computational semantics of programming languages.

In the computational (also called operational or interpretative) approach to formal semantics of programming languages, the meaning of programs is given by showing how their computations proceed for any acceptable input data. This is done by defining (not necessarily

explicitly) an abstract computer and its next-state mapping. Programs are usually incorporated within the states of that abstract computer. Thus, we need two sets of transition rules, one describing the translation of programs and data into initial states of the computer, and the other describing the next-state mapping.

In order to appreciate the advantages of linked trees, we will briefly look at the historical development of data structures used in models for computational semantics. The early models for computational semantics [1, 5] used strings to represent both programs and intermediate states of the abstract computer. In these models the transition rules, i.e. the rules for manipulation of strings, were described by generalized Markov algorithms. The generalization of Markov algorithms actually allowed some implicit use of tree structures. In [12] push down store was used.

Derivation trees of context-free grammars (more precisely, labelled, ordered, rooted trees) was the basic data structure used by Knuth in [6]. When interpreting Knuth's model in terms of an abstract computer, the initial translation consists of the parsing of a given program, whereas the next-state transitions are given by rules for manipulating the values of the attributes associated with nonterminals, i.e. with the nodes of a derivation tree. In this model, however, the structure of the tree remains unchanged throughout a computation.

A very useful for computational semantics is the notion of abstract syntax tree introduced by McCarthy in [8]. Abstract syntax trees are more explicitly related to semantics than derivation trees. Typically, a branching node of an abstract syntax tree is labelled by an

operator. Each of its sons is then a root of a subtree describing one operand. The leaves are labelled by simple operands. In comparison to the derivation trees, the abstract syntax trees allow us to neglect the unessential syntactic properties (so-called syntactic sugar), and to concentrate on the essential structural properties.

An example of an Algol-like derivation tree is in Figure 1.1, while the corresponding abstract syntax tree is given in Figure 1.3. The same abstract syntax tree may also correspond to the derivation tree of Figure 1.2.

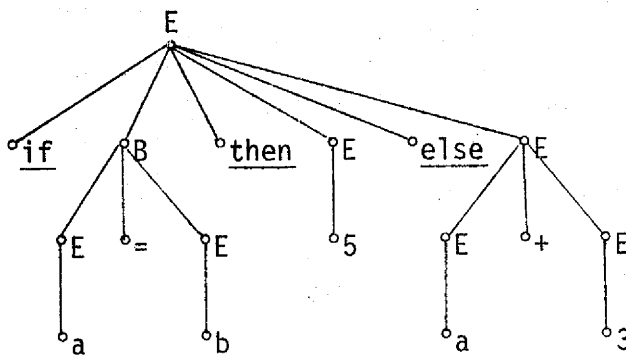


Figure 1.1 Derivation tree for the expression
if a=b then 5 else a+3

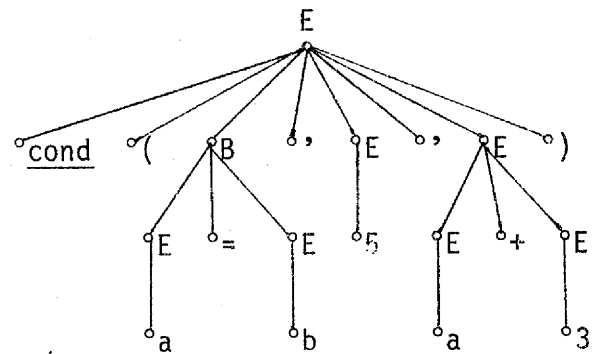


Figure 1.2 Derivation tree for the expression
cond(a=b, 5, a+3)

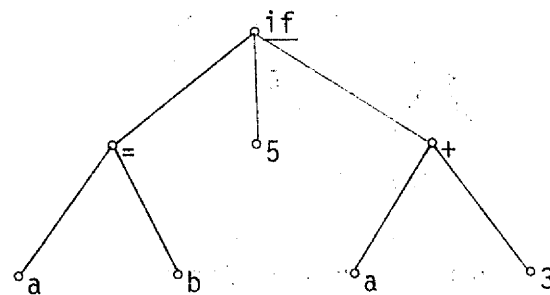


Figure 1.3 Abstract syntax tree for both expressions
if a=b then 5 else a+3 and cond(a=b, 5, a+3)

A model for computational semantics based on McCarthy's abstract syntax trees is the Vienna definition method [7, 11] which is a parent of the model given in [2] and studied in this paper. The use of trees as the basic data structure in the Vienna definition method allows an easy treatment of the context-free aspects of programming languages. However, the non-context-free properties, such as scope of variables, declarations, etc., cause more difficulties.

Trees with pointers, called linked trees [2], form a new data structure type which allows the explicit showing of all structural properties of a programming language. For example, the association of variable use with variable declaration is performed by a pointer from the point of use to the point of declaration. As tools for the formal description of transformations on linked trees, and more generally on linked forests, we define linked-forest manipulation systems (l.f.m.s.) which describe these transformations by means of production schemas. The notion of l.f.m.s. is a generalization of the notion of subtree replacement systems introduced by Rosen in [9].

In Section 2 we informally describe Rosen's subtree replacement systems and their generalization to linked-forest manipulation systems. Next, we discuss the model for computational semantics which is based on l.f.m.s., and an Algol-like programming language ALG on which this model is demonstrated. The formal definition of l.f.m.s. and that of the model for formal description of programming languages are given in Sections 4 and 5, respectively. We conclude with examples from the language ALG and a formal description of the λ -calculus.

2. Tree and Linked Forest Manipulation Systems

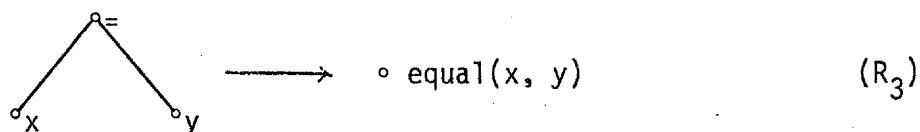
2.1 Subtree Replacement Systems

In [9] a formalism is given for defining systems which manipulate trees, i.e. which describe transformations on rooted, ordered, labelled trees. Such systems consist mainly of rules for subtree replacement. For example a rule such as

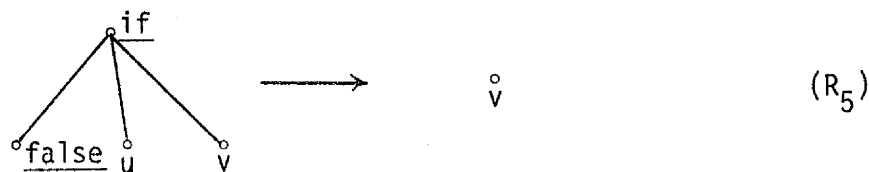
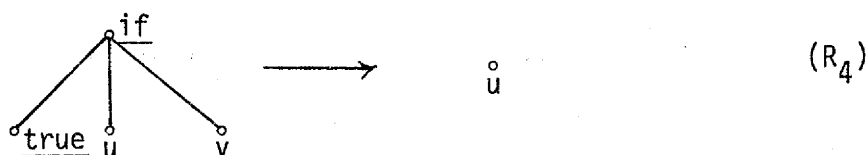
$$\begin{array}{c} \circ \\ \swarrow \quad \searrow \\ \circ x \quad \circ y \end{array} \quad \longrightarrow \quad \circ \text{mult}(x, y) \quad (R_1)$$

indicates that if in the manipulated tree there is a subtree similar to the left hand side of the rule, i.e. having the same structure, but with integer numbers in place of x and y , then this subtree can be replaced by the subtree consisting of a single node labelled with the number resulting from the multiplication of these two numbers. In this rule R_1 two label parameters, x and y , are used. Such parameters have a domain which is, in this case, the set of integers. A function name, mult , is also used in a functional denotation together with the label parameters x and y . This function name designates a function which is defined in the system; in this case, it denotes the multiplication of integers. The functional denotation $\text{mult}(x, y)$ designates the result of multiplying an integer number x by an integer number y . To illustrate tree manipulation systems with an example we give a few other rules.

$$\begin{array}{c} \circ \\ \swarrow \quad \searrow \\ \circ x \quad \circ y \end{array} \quad \longrightarrow \quad \circ \text{minus}(x, y) \quad (R_2)$$



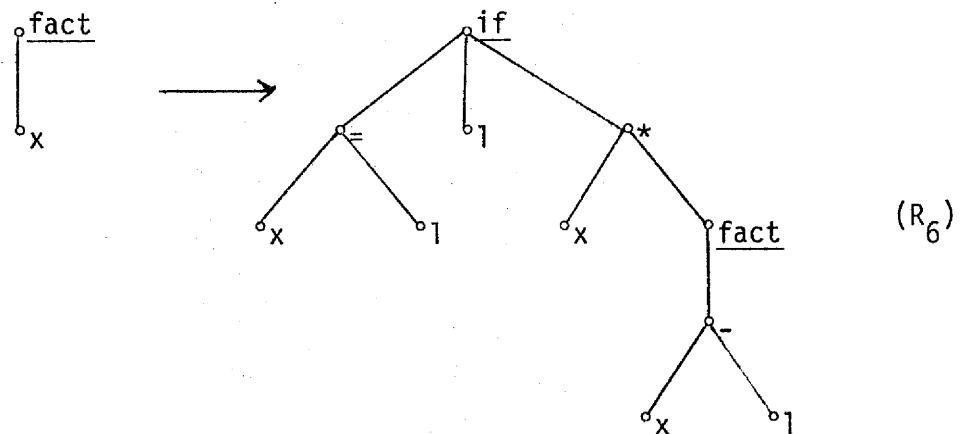
Rule R_2 is similar to R_1 except that the operation is subtraction of integers instead of multiplication. Rule R_3 is also similar to R_1 , but here `equal` is the name of the logical function which takes two integers as arguments and yields true if both arguments are equal, false otherwise. The three rules R_1 , R_2 and R_3 can be considered as definitions of binary operators $*$, $-$ and $=$. The definition of a ternary operator, namely the conditional operator, is given by the two rules that follow.



In rules R_4 and R_5 two parameters u and v are used and they are called tree parameters as their domains are trees. Rule R_4 indicates that if a subtree of the tree we are manipulating is similar to the right hand side of R_4 (with u and v being replaced by some subtrees), then we can replace that subtree by subtree u . Since u is supposed to be the abstract tree representation of then-expression and

and v that of the else-expression of a certain conditional expression, R_4 indicates that if the condition is true then the expression u is evaluated. On the other hand, R_5 indicates that if the condition is false then v is evaluated.

As a unary operator, the factorial function can be defined recursively using the operators $*$, $-$, $=$ and if by means of the following rule.



For instance, we can use the above rules $R_1 - R_6$ to compute factorial 2. This is shown in Figure 2.1. Each step of the computation is performed by means of one of these rules which is indicated above the double arrow. Note that this computation is not unique, but all terminating computations (using the above rules) lead to the same answer.

Note that the application of the rules is done in any order and that a rule can be applied any number of times. Also, each rule specifies the replacement of some subtree by another subtree. Furthermore, any parameter can occur at most once on the left hand side of a rule. A generalization of subtree manipulation systems is defined next. In this

generalization the restrictions mentioned above are removed, and new features are introduced.

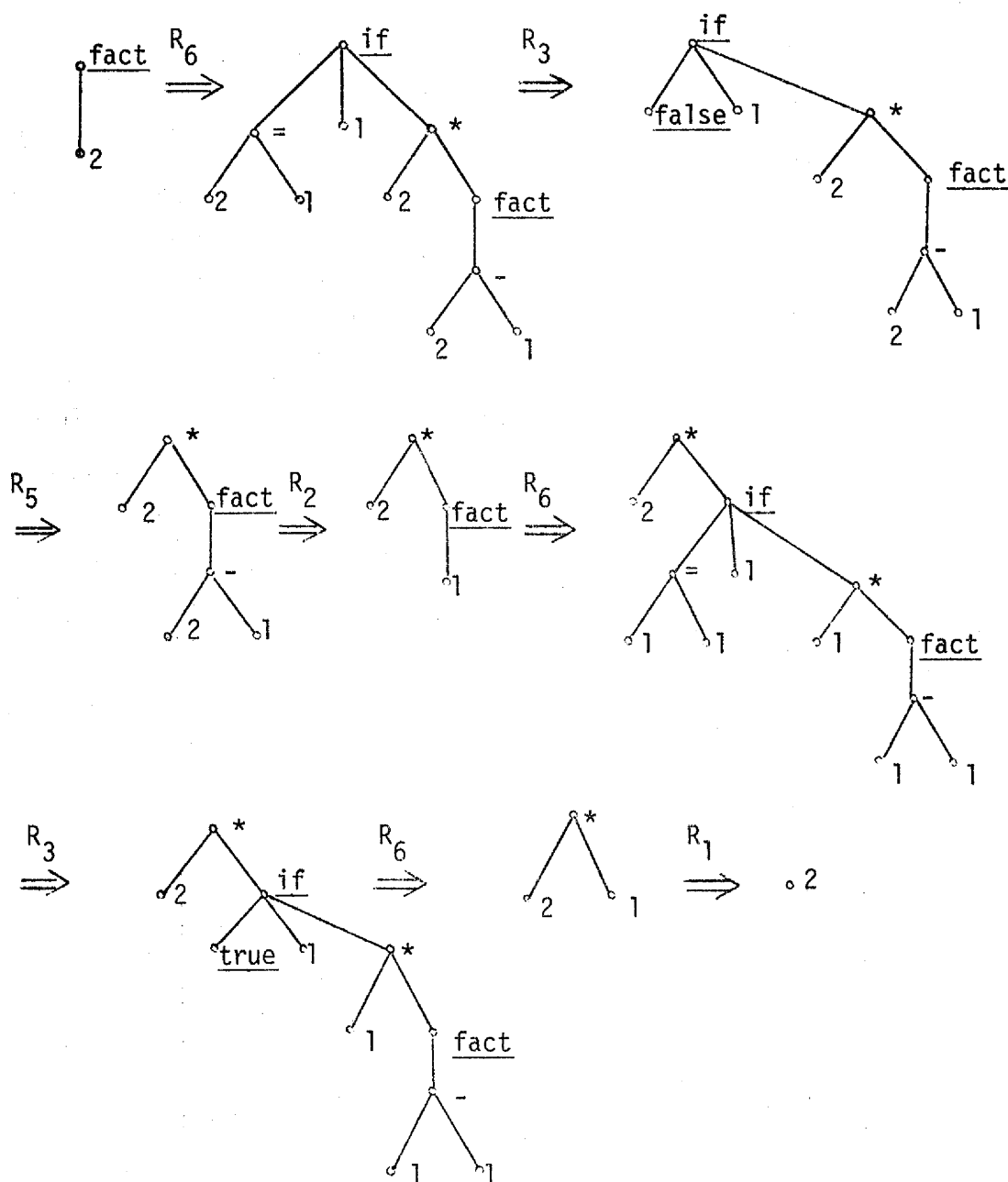


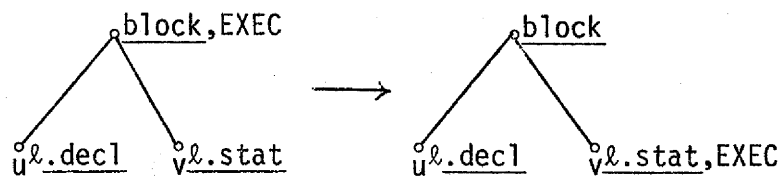
Figure 2.1 Computation of $2!$ using rules R_1 - R_6

2.2 Linked Forest Manipulation Systems

A linked forest manipulation system (l.f.m.s.) essentially consists of a set of rules or production schemas which define transformations on linked forest, i.e. on sets of trees with pointers (links) between them. Here, the trees are rooted, ordered and multilabelled, i.e. a node of the tree may have several labels attached to it. However, not all the labels must be shown in the production schemas. For example, suppose we are describing the semantics of a block in Algol. A block consists of a list of declarations followed by a list of statements. The execution of a block should start at the beginning of the list of statements. Suppose that the tree representation of a block has a root node labelled block, this root node having two sons, the left one labelled l. decl and the right one labelled l. stat. A control label, EXEC, is assumed to control the execution of the linked tree representation of an Algol program. The rule which indicates that the execution of a block starts with the list of statements is the following:



The left hand side of the production schema (P₁) need not match a subtree so as to use the production. It is sufficient that it matches a substructure (subgraph) of the tree representing the program. To describe the same property using a subtree replacement system, and assuming we can have multilabelling, one would have to give the following rule

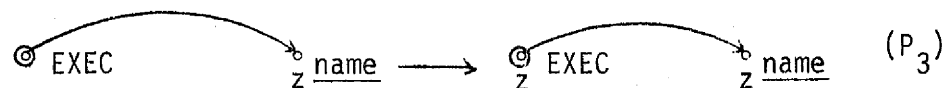


Here we have to use parameters u, v and additional labels to describe a complete subtree most of which remains unchanged.

A simple and illustrative use of links can be given for describing the semantics of the goto-statement. Assuming that a link is constructed from the node labelled goto to the node representing the statement to which the jump should be made, the following production defines the semantics of the goto-statement.



When we need to be precise about the descendants of a node in a production, we use a distinguished sort of node called pivotal node (graphically distinguished by a double circle). For instance, if the production changes the structure of the subtree rooted at a certain node, that node should be represented by a pivotal node. This pivotal node indicates that in the matching process the subtree under the pivotal node should be matched with the root of a complete subtree and not any substructure. Consider the following production schema which partially describes the copy rule for a function call in Algol.



Production P_3 indicates that whenever a formal parameter called by $name$ is encountered in a procedure body, with the corresponding actual

parameter being an expression represented by linked tree z , then z is copied at the execution point. All pointers from the nodes of z to some other nodes are also copied. Figure 2.2 gives the relevant part of a derivation based on production P_3 .

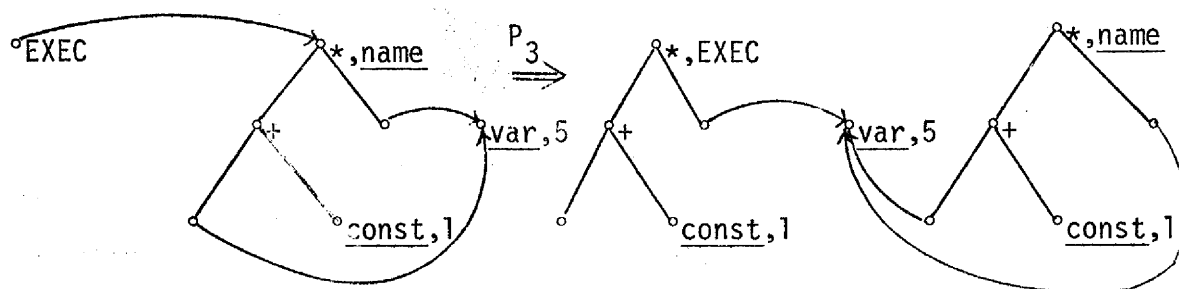
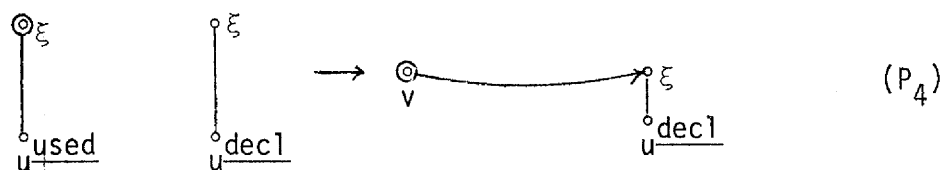


Figure 2.2 An illustration of production P_3

In any production schema of a l.f.m.s. it is possible to have any parameter occur more than once on its right hand side. The following rule is an illustration of this possibility and describes the matching of declarations of variables with their use. A link is also constructed from the point of use to the declaration.



One additional feature of the l.f.m.s. is the mechanism allowing the forcing of a certain order of application of the production schemas. Every production in a l.f.m.s. is labelled and has success and failure fields similar to the ones in Rosenkrantz's Programmed Grammars [10] or in the SNOBOL programming language. The labels of the success

field indicate which productions may be used next if the considered production is applied successfully. Those of the failure fields are the labels of the productions which may be used next in case the considered production cannot be applied. Also, in any $\lambda.f.m.s.$ there should be some production labelled START, and at least one production should have the label STOP in its success or failure field. Any derivation in a $\lambda.f.m.s.$ starts with the application of a production labelled START, and ends with either the success in applying a production whose success field consists of the label STOP, or the failure to apply a production whose failure field consists of STOP. Sometimes a label ERROR is used in either a success or a failure field to indicate a special type of termination.

Figure 2.3 gives a $\lambda.f.m.s.$ which can be used to match the used variables of a block with their declaration connecting them with a pointer (production labelled START). Then it removes the names of those variables which have become superfluous (production labelled L1).

Production Label	Production Schema	Success Field	Failure Field
START		START	L1
L1		L1	STOP

Figure 2.3 Example of an $\lambda.f.m.s.$

3. Describing a Programming Language Formally

Linked forest manipulation systems can be used to define completely and formally the computational semantics of programming languages. As has been discussed in the introduction, the computational definition of semantics includes the translation of programs into an initial state of an abstract computer. This will be done by the so-called syntax part of the model that we are giving. The syntax will be described by the usual context-free grammar of the language, plus a formal description of the non-context-free syntactical restrictions (the checking of these restrictions being usually, and incorrectly, called semantical analysis). The syntax part also describes the translation of a program into an abstract linked tree which represents the initial state of the abstract computer. The result of the syntax processing is then manipulated by the semantics part which is one l.f.m.s. It essentially produces a linked forest representing the results computed by the given program. The abstract computer, which is not explicitly defined, has the linked forests as states and the l.f.m.s. as the next state mapping. The linked forest obtained when a computation halts can be mapped into a desired form of "output data".

Before formally describing how this is done we will explain in this section the approach on a specific programming language ALG. This language is essentially ALGOL 60, i.e. it has all the features of ALGOL 60 which are difficult to describe, particularly parameter passing to procedures and functions. However, ALG does not have arrays, reals and other features which would not cause any difficulties while making the

language too large. (For the description of full ALGOL 60 by this method see [12].)

3.1 The Syntax Part

The syntax of ALG is given in Table I. It consists of a set of rules each formed from a context-free production, an abstract syntax tree denotation, and possibly a l.f.m.s. which describes non-context-free properties. A rule in the syntax part defines, partially or totally, a non-terminal; and associates a set of abstract syntax trees (and more generally a linked tree) with this non-terminal. Thus, rule 1 defines the non-terminal `<ident>` as any element of a set ID of identifiers; it also associates with `<ident>` the set of all single node trees labelled by an identifier. Similarly for `<const>` in rule 2, where INT is the set of integers plus a nil element. Rule 3 defines `<var>` and associates with it the set of two node trees such that the root node is labelled with an identifier and the other node labelled with {int, used}. The non-terminal `<prim>` is defined by rules 4-7. In rule 4 it is partially defined as being a variable and is associated with any tree which is associated with `<var>`. Similarly for rules 5 and 6. Rule 7 indicates that a primary can be a function call, that is `<prim>` can be an identifier followed by a list of actual parameters, `<l.a.p.>`, between brackets. The set of trees associated with this partial definition of `<prim>` consists of all trees obtained from the trees associated with `<l.a.p.>` by adding the labels call, func and any identifier to their root nodes. Rules 8-10 define `<factor>` recursively. Rule 8 partially defines it as a primary and associates

with it all trees associated with $\langle \text{prim} \rangle$. Rule 9, however, associates with $\langle \text{factor} \rangle$ any tree consisting of a root node labelled $*$ which has two sons, the left one being the root of a subtree which can be any tree associated with $\langle \text{factor} \rangle$, and the right one being the root of a subtree which can be any tree associated with $\langle \text{prim} \rangle$. Similarly for rule 10. The reader can easily interpret rules 11-22 by analogy with the first 10 rules. Rule 23, while partially defining $\langle \text{stat} \rangle$ associates (in a recursive manner) with non-terminal $\langle \text{stat} \rangle$ a set of trees such that each is formed of two trees by joining their root nodes. The left tree is of the form $\begin{array}{c} \circ \xi \\ | \\ \text{label, decl} \end{array}$ for some $\xi \in \text{ID}$, and the right tree is any tree associated with $\langle \text{stat} \rangle$.

While rules 1-27 have their l.f.m.s.'s empty, rule 28 which defines $\langle \text{block} \rangle$ has a non-empty l.f.m.s. This implies that the set of trees associated with $\langle \text{block} \rangle$ is not simply the set of trees indicated by the abstract syntax tree denotation of that rule, but the set of trees resulting from that set by manipulating every tree by the given l.f.m.s. The first production (labelled START) in the l.f.m.s. checks for multiple declarations of a variable in the block. The second production (labelled L1) matches the declaration of every variable with its use in the block. Production L2 checks for the use of a variable with the wrong type, in particular it checks for the declaration of a label as a variable. Finally production L3 simplifies the declaration subtrees of the variables of the block by removing the names which have become superfluous and reducing them to single nodes.

Rules 29-36 can be easily understood by the reader. Rule 37 has a non-empty $\ell.f.m.s.$ and partially defines a list of actual parameters $\langle \ell.a.p. \rangle$. The $\ell.f.m.s.$ of this rule integrates the rightmost actual parameter $\langle a.p. \rangle$ into the list of actual parameters. An operator $\&$ is used which concatenates two trees by joining their roots. Similarly, in rule 41 the $\ell.f.m.s.$ integrates the rightmost formal parameter $\langle f.p. \rangle$ into the list of formal parameters $\langle \ell.f.p. \rangle$. Rules 42 and 43 define $\langle decl \rangle$, the first one for variables and the second one for procedures and functions. The label nil represents the initial undefined value that a variable or a function has. The $\ell.f.m.s.$ of rule 43 deals with procedure and function declarations. Its first three productions and the fifth one are similar to those of the $\ell.f.m.s.$ for $\langle block \rangle$ in rule 28, while the fourth and the sixth (labelled L3 and L5) are used for technical reasons. The non-terminal $\langle \ell.decl \rangle$ is defined in rules 44 and 45, while $\langle program \rangle$ is defined in 46 and its $\ell.f.m.s.$ checks for variables used in the program but declared nowhere. The set of trees associated with $\langle program \rangle$ consists of all the trees associated with $\langle block \rangle$ which have no variable which is used but undeclared, the control word EXEC labelling their root nodes.

3.2 The Semantics Part

The semantics of ALG is given in Table II. It consists of one $\ell.f.m.s.$ which gets a linked tree associated with $\langle program \rangle$ from the syntax part (as initial value), and produces a linked tree representing the results of executing the program. The first two productions indicate that the execution starts with the list of statements of the

block forming the program, and ends with the end of execution of this list of statements. The execution of a list of statements from left to right is described by productions 3-5. The semantics of the goto-statement is given by production 6, while productions 7 and 8 give the semantics of the assignment statement. The evaluation of arithmetic expressions is described in productions 9-13, and that of the conditional expressions and conditional statements is described in productions 18-23.

Productions 24-27 procedure and functions calls. In production 24 the lists of formal and actual parameters are superposed using the operator \oplus which operates on trees with similar structure. The addition of label & stat is only for technical reasons, for evaluating the actual parameters from left to right. Also, a copy of the body of the procedure is made at the calling point. The execution starts however with the evaluation of the actual parameters. Production 29 indicates that after this evaluation the copy of the procedure body is executed. Productions 30 and 31 describe the return from procedure and function calls; a value is returned in the case of a function call.

The passing of parameters to the procedure call by value is defined by production 25, while the passing by name is defined by productions 26-28.

In this section we have tried to give an informal description of the way a programming language can be formally defined to help the reader in understanding the formal definitions of l.f.m.s. and that of language description systems given in the next two sections. The reader could however skip these two sections and go to Section 6 in

which a detailed example of a program in ALG is treated. The two operators $\&$ and \oplus used in the description of ALG are defined algebraically in the next section, but they can as well be defined by an elementary l.f.m.s. as shown in [2].

4. Formalism for Linked Forest Manipulation Systems

4.1 Trees and Forests

Given any set V , called vocabulary, we define the set of tree expressions, or simply trees, over V as follows.

Definition 4.1.1

- (i) Every finite subset A of V is a tree expression;
- (ii) If A is a finite subset of V and $\alpha_1, \dots, \alpha_n$ are tree expressions then $A[\alpha_1, \dots, \alpha_n]$ is a tree expression.

□

Example 4.1.1 Let $V = \{a, b, c, d, e, f\}$. The following are tree expressions, or simply trees, over V .

$$\alpha = \{a, b\}$$

$$\beta = \{a\}[\{b, a\}, \{b, c\}[\{d\}, \{e, f\}], \{a, c\}]$$

The notion of equivalence of tree expressions, or equality of represented trees, is an important part of the definition of trees and is given below.

Definition 4.1.2 Two trees α and β are equal if either

- (i) α and β are finite subsets of V and are equal (as sets) or
- (ii) $\alpha = A[\alpha_1, \dots, \alpha_n]$, $\beta = B[\beta_1, \dots, \beta_n]$, $A = B$ and α_i equals β_i for all i , $1 \leq i \leq n$.

□

As a result of this definition the trees $A[\alpha_1, \alpha_2]$ and $A[\alpha_2, \alpha_1]$ are not equal. More generally, the order of the tree

expressions $\alpha_1, \dots, \alpha_n$ in $A[\alpha_1, \dots, \alpha_n]$ is crucial and $[\alpha_1, \dots, \alpha_n]$ can be regarded as an n -tuple of tree expressions.

Definition 4.1.3 A tree expression α is said to be a subtree expression, or simply subtree, of tree expression β , if either

- (i) α is equal to β
- or (ii) $\beta = B[\beta_1, \dots, \beta_n]$ and α is a subtree of β_i for some i , $1 \leq i \leq n$.

□

Example 4.1.2 Referring to trees α and β of Example 2.1, we have that

$\{a, b\}$ is a subtree of α

and $\{b, c\}[\{d\}, \{e, f\}]$ is a subtree of β .

Moreover, α is a subtree of β .

Definition 4.1.4 Any finite subset of trees over vocabulary V is called a forest over V .

□

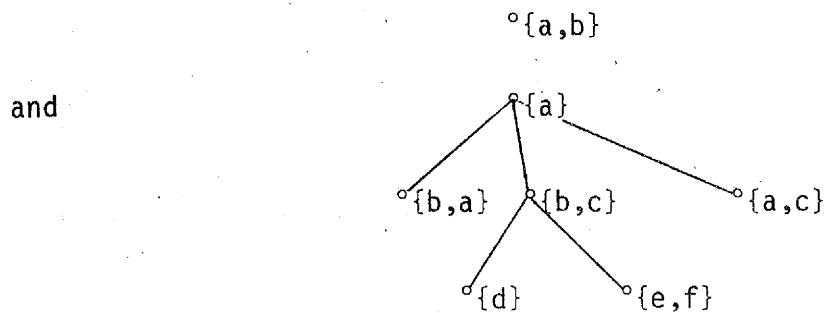
Trees are represented graphically using nodes and edges. The nodes are drawn as small circles and are labelled by a finite subset of V , while edges are lines joining two nodes. The graphical representation (g.r.) of a tree α is constructed as follows. If α is a finite subset of V , say A , then

${}^\circ A$

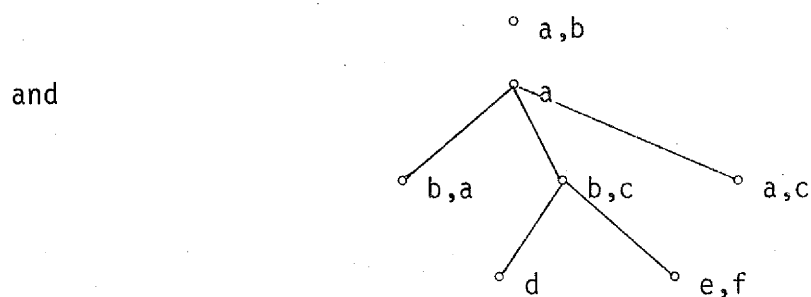
is the graphical representation of α and the node labelled with A is called the root node of α . However, if α is the tree

$A[\alpha_1, \dots, \alpha_n]$ then the g.r. of α is obtained from the g.r.'s of $\alpha_1, \dots, \alpha_n$ by constructing a new node labelled A , called the root node of α , and constructing edges from that root node to each of the root nodes of the g.r.'s of $\alpha_1, \dots, \alpha_n$; the order of the trees $\alpha_1, \dots, \alpha_n$ should be respected.

Example 4.1.3 Trees α and β given in Example 2.1 are respectively represented graphically by



In general the set brackets are removed so that the representations would be respectively



The class of trees that we have defined here corresponds to what is commonly known as the class of rooted ordered multilabelled trees. We denote by $T_0(V)$ the set of all such trees over vocabulary V , and by $F_0(V)$ the set of all forests over vocabulary V .

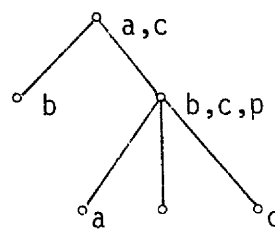
4.2 Trees and Forests with a Pivotal Node

Let p be a symbol not occurring in V . We define the class of trees with a pivotal node over V , and denote it by $T_p(V)$, as the subset of $T_0(V \cup \{p\})$ in which p labels exactly one node. Also, the class of forests with a pivotal node over V , denoted by $F_p(V)$, is defined as being the class of all finite subsets of $T_0(V \cup \{p\})$ such that exactly one tree in the subset is a tree in $T_p(V)$.

The use of these classes of trees and forests will become clear when defining the notion of linked forest manipulation system.

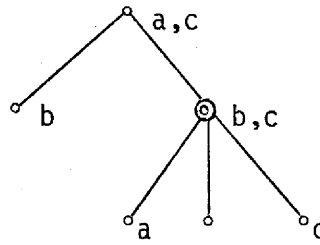
The set of all trees, with and without a pivotal node is denoted $T(V)$, i.e. $T(V) = T_0(V) \cup T_p(V)$. Similarly, the set of all forests, with and without a pivotal node is denoted $F(V)$, i.e. $F(V) = F_0(V) \cup F_p(V)$.

Example 4.2.1 Let $V = \{a, b, c\}$ and let α be the tree represented graphically by



α is a tree over V with a pivotal node.

Graphically however, the label p is replaced by a double circle around its corresponding node; e.g. the tree given above would be represented by



4.3 Operations on Trees and Forests

4.3.1 Concatenation and Superposition of Trees

Two binary operations on trees without a pivotal node are defined in what follows. The first one corresponds to the concatenation of two trees and it is denoted $\&$, while the second one corresponds to the superposition or merging of two trees and is denoted \oplus .

$$\& : T_0(V) \times T_0(V) \rightarrow T_0(V)$$

such that for any α and β in $T_0(V)$, say $\alpha = A[\alpha_1, \dots, \alpha_m]$ and $\beta = B[\beta_1, \dots, \beta_n]$ for some $m, n \geq 0$ (we assume that $A = A[\]$)

$$\alpha \& \beta = A \cup B[\alpha_1, \dots, \alpha_m, \beta_1, \dots, \beta_n] .$$

Note that $\&$ is associative but is not commutative. Thus, we will write $\alpha \& \beta \& \gamma$ to mean either $\alpha \& (\beta \& \gamma)$ or $(\alpha \& \beta) \& \gamma$

$$\oplus : T_0(V) \times T_0(V) \rightarrow T_0(V)$$

such that for A and B finite subsets of V ,

$$(i) \quad A \oplus B = A \cup B$$

$$(ii) \quad \text{if } \alpha = A[\alpha_1, \dots, \alpha_m] \text{ then}$$

$$B \oplus \alpha = \alpha \oplus B = A \cup B[\alpha_1, \dots, \alpha_m]$$

(iii) if $\alpha = A[\alpha_1, \dots, \alpha_m]$ and $\beta = B[\beta_1, \dots, \beta_n]$ with
 $n \geq m \geq 1$ then

$$\beta \oplus \alpha = \alpha \oplus \beta = A \cup B[\alpha_1 \oplus \beta_1, \dots, \alpha_m \oplus \beta_m, \beta_{m+1}, \dots, \beta_n].$$

Note that \oplus is an associative and commutative operation.

This operation corresponds to the superposition of two trees with left alignment. Note that both $\&$ and \oplus can be extended to functions from $T_0(V) \times T_p(V) \cup T_p(V) \times T_0(V)$ into $T(V)$.

4.3.2 Skeletal Operator on Trees

The skeletal operator π is defined for any tree over V as follows.

$$\pi : T(V) \rightarrow T(V)$$

such that for any finite subset A of $V \cup \{p\}$

$$(i) \quad \pi(A) = A - \{p\}$$

(ii) if $\alpha = A[\alpha_1, \dots, \alpha_m]$ then

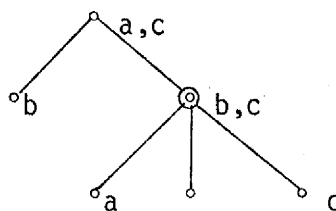
$$\pi(\alpha) = \begin{cases} \pi(A) \\ A[\pi(\alpha_1), \dots, \pi(\alpha_m)] \text{ otherwise.} \end{cases}$$

Thus, the effect of π is to remove the subtree whose root is a pivotal node, except for the root, and remove p from the set of labels of that root.

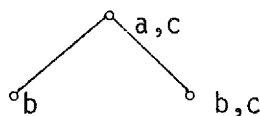
Definition 4.3.2.1 We define the skeleton of a tree $\alpha \in T(V)$ as being the tree $\beta \in T_0(V)$ obtained from $\pi(\alpha)$ by the replacement of every set of labels by the empty set.

□

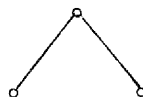
Example 4.3.1 Let $V = \{a, b, c\}$ and let α be the tree represented by



$\pi(\alpha)$ would be



and the skeleton of α would be



Definition 4.3.2.2 Two forests Φ and Ψ in $F(V)$ have the same skeleton if there exists a bijection $b : \Phi \rightarrow \Psi$ such that for every tree $\alpha \in \Phi$, $b(\alpha)$ has the same skeleton as α .

□

4.3.3 Subtree Replacement

Given $\tau, \alpha, \beta \in T_0(V)$ such that α is a subtree of tree

τ , the replacement of α by β in τ , denoted τ_{β}^{α} , is well defined; and the result of the replacement is a tree, i.e. $\tau_{\beta}^{\alpha} \in T_0(V)$.

We can easily generalize this subtree replacement to forests. Thus, if $\Phi \in F_0(V)$, $\alpha, \beta \in T_0(V)$ and α is a subtree of Φ , i.e. a subtree of a tree $\tau \in \Phi$, then the replacement of α by β in Φ , denoted Φ_{β}^{α} , is defined as the forest resulting from Φ if τ is replaced by τ_{β}^{α} .

4.3.4 Substructure Replacement

4.3.4.1 Substructures in Trees and Forests

The notion of substructure in a tree or forest is a generalization of the notion of subtree in a tree or forest. First, we have to define the notion of trim of a forest.

Definition 4.3.4.1a Let $\Phi, \Psi \in F_0(V)$; we say that Φ is a trim of Ψ if Φ is obtained from Ψ by replacing some subtrees of Ψ by the empty tree.

□

Definition 4.3.4.1b Let $\Phi \in F_0(V)$, $\alpha \in T_0(V)$ and $\beta \in T(V)$. We say that α is a substructure of Φ corresponding to β if α is a subtree in some trim of Φ , has the same skeleton as β , and, in case $\beta \in T_p(V)$, α is a subtree of Φ .

□

We denote by $\text{sub}_\alpha(\Phi)$ the set of all substructures of Φ corresponding to α . For trees the notions of trim and substructure is the same as for singleton forests, i.e. forests with a single tree.

We will further generalize the notion of substructure to that of substructures corresponding to a forest.

Definition 4.3.4.1c Let $\Phi \in F_0(V)$, and $\Psi \in F(V)$ with $\Psi = \{\beta_1, \dots, \beta_n\}$ and $\alpha_i \in T(V)$ for all i , $1 \leq i \leq n$. For any $\Gamma \in F_0(V)$, $\Gamma = \{\alpha_1, \dots, \alpha_n\}$, we say that Γ is a substructure of Φ corresponding to Ψ if

- (i) $\forall i, 1 \leq i \leq n, \alpha_i \in \text{sub}_{\beta_i}(\Phi)$
- (ii) $\forall i, \forall j$ such that $i \neq j$, α_i and α_j have no common nodes
- and (iii) if for some k , β_k has a pivotal node then α_k is a subtree of Φ .

□

4.3.5 Replacement of Substructures in Forests

Let $\Phi \in F_0(V)$, $\alpha \in T_0(V)$ and $\beta \in T(V)$ such that α is a substructure of Φ corresponding to β . We define the replacement of α by β in Φ , denoted Φ_β^α as follows.

- (i) If β has a pivotal node, α would be a subtree of Φ and Φ_β^α is defined as the replacement, in Φ , of subtree α by tree β . Notice that in this case Φ_β^α would be in $F_p(V)$ although Φ is in $F_0(V)$.

- (ii) If β has no pivotal node then α and β have the same skeleton, i.e. the same underlying structure, and Φ_{β}^{α} is defined as the replacement, in Φ , of the labels at a node of α by the labels at the corresponding node (in the skeletal correspondence) of β .

Now, we can define the replacement, in Φ , of a substructure $\Gamma \in F_0(V)$ of Φ corresponding to forest $\Psi \in F(V)$ by Ψ as follows.

If $\Gamma = \{\alpha_1, \dots, \alpha_n\}$ and $\Psi = \{\beta_1, \dots, \beta_n\}$ then, assuming that $\beta_i \in T_0(V)$ for all $1 \leq i \leq n-1$ and $\beta_n \in T(V)$, we define that replacement by

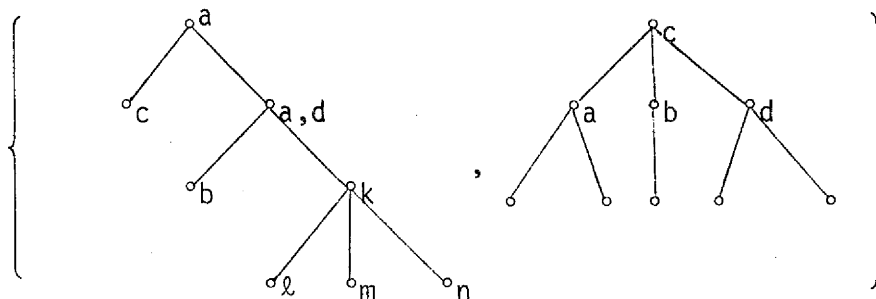
$$\Phi_{\Psi}^{\Gamma} = \left(\dots \left(\left(\Phi_{\beta_1}^{\alpha_1} \right)^{\alpha_2} \right) \dots \right)^{\alpha_n}_{\beta_n}.$$

These successive substructure replacements are well defined because: (i) α_i and α_j , for $i \neq j$, have no common nodes; and (ii) as $\beta_i \in T_0(V)$ for all $1 \leq i \leq n-1$ then

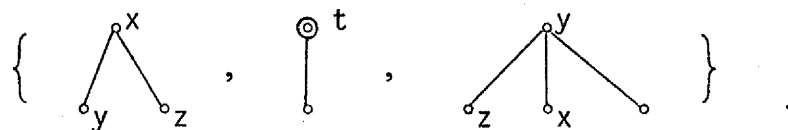
$$\left(\dots \left(\Phi_{\beta_1}^{\alpha_1} \right)^{\alpha_2} \dots \right)^{\alpha_{n-1}}_{\beta_{n-1}} \in F_0(V).$$

Example 4.3.1

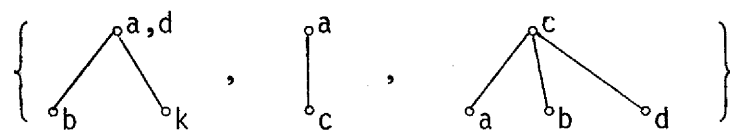
Let Φ be the forest



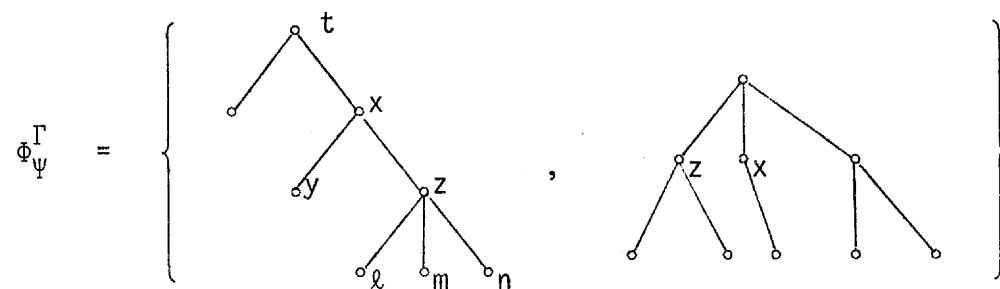
and let Ψ be the forest



Forest Γ which is equal to



is a substructure of Φ corresponding to Ψ . The replacement of Γ by Ψ in Φ would be



4.4 Linked Forests; Label, Pointer and Tree Parameters

4.4.1 Linked Forests

Let $\vec{\Delta} = \{\vec{i} \mid i \in N\}$, $\overleftarrow{\Delta} = \{\overleftarrow{i} \mid i \in N\}$, and $\Delta = \vec{\Delta} \cup \overleftarrow{\Delta}$.

Assume that V is a vocabulary such that $V \cap \Delta = \emptyset$. Any element of Δ is called a pointer label and any pair $\langle \vec{i}, \overleftarrow{i} \rangle$ for some $i \in N$ is called a pointer (or link). $F(V \cup \Delta)$ would denote the set of all forests over $V \cup \Delta$. The notion of linked forest is defined below.

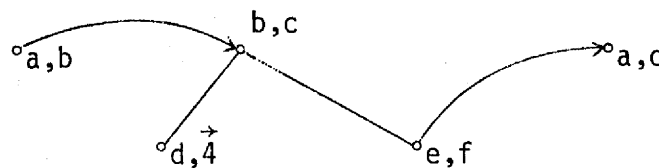
Definition 4.4.1 Any forest $\Phi \in F(V \cup \Delta)$ is said to be a linked forest over V if no pointer label appears at more than one node of Φ .

□

We denote by $F^L(V)$ the set of all linked forests over V . Also we distinguish between the set of linked forests without pivotal node and the set of linked forests with a pivotal node. The former is denoted $F_0^L(V)$ and the latter $F_p^L(V)$. Thus, $F^L(V) = F_0^L(V) \cup F_p^L(V)$. Note that a trim of a linked forest is a linked forest. However, the forest obtained by replacing, in a linked forest, a substructure by its corresponding forest is not necessarily a linked forest.

Example 4.4 The following is a linked forest over $V = \{a, b, c, d, e, f\}$ and consists of three trees.

$\Phi = \{\{a, b, \vec{1}\}, \{b, c, \vec{1}\}, \{d, \vec{4}\}, \{e, f, \vec{3}\}, \{a, c, \vec{3}\}\}$. It will be represented graphically as



Notice that the nodes labelled by \vec{i} and \overleftarrow{i} , for some $i \in N$, are linked by a pointer whose origin is at the node labelled with \vec{i} and whose end is at the node labelled with \overleftarrow{i} . However, if either \vec{i} or \overleftarrow{i} , for some i , is not used while the other one is used, then the one which is used is treated as any other label.

4.4.2 Label, Pointer and Tree Parameters

In linked forests over vocabulary V there are mainly three sorts of objects. There are the labels, elements of V , the pointer labels, elements of Δ , and the tree or subtrees over $V \cup \Delta$. We would like to have some kind of variables over these three domains, namely over V , Δ and $T(V \cup \Delta)$, so as to describe linked forests using these variables as labels. Such a linked forest would represent many, and in general an infinity, of linked forests over V . A variable having for domain a subset of V will be called a label parameter. A variable having for domain a subset of Δ will be called a pointer parameter. A variable having for domain a subset of $T(V \cup \Delta)$ will be called a tree parameter.

A label parameter can have for domain any subset of V , while tree parameters will be allowed to have domains of the form $T_0(W \cup \Delta)$ for some $W \subseteq V$.

However, pointer parameters are of two kinds. Those that have for domain $\vec{\Delta}$ and those that have for domain $\overleftarrow{\Delta}$. Thus, if we denote by C the set of pointer parameters then $C = \vec{C} \cup \overleftarrow{C}$ where \vec{C} consists of the pointer parameters, such as \vec{a} , whose domain is $\vec{\Delta}$, and \overleftarrow{C} consists of the pointer parameters, such as \overleftarrow{a} , whose domain is $\overleftarrow{\Delta}$.

4.4.3 Substitution of Parameters

The use of parameters in linked forests would presumably lead to the substitution of some element of their domain for their occurrences. For any parameter x whose domain is D_x , any $n \in D_x$ and any linked forest Φ using parameter x , we denote by $\Phi(x \leftarrow n)$ the result of

substituting in Φ , n for every occurrence of x . That is, n would replace every occurrence of x in every tree of Φ . In general, for k parameters x_1, \dots, x_k whose respective domains are D_{x_1}, \dots, D_{x_k} , and for $n_i \in D_{x_i}$ we denote by $\Phi(x_i \leftarrow n_i \mid 1 \leq i \leq k)$ the result of substituting n_i for each occurrence of x_i in Φ .

4.4.4 Functional Denotations

To manipulate parameters over some domain, and sometimes combine them, we need functions. In particular for tree parameters, we have defined two functions namely $\&$ and \oplus . Others can be defined when needed. Thus, a set of basic functions should be given for each type of parameters that need to be manipulated.

In general, given a set A and a set of function names over A (i.e. a set such that for any $f \in F$, f is a function from A^n to A for some n), let $\{x_1, \dots, x_\ell\}$ be the set of variables over V and $D_{x_1}, \dots, D_{x_\ell}$ their respective domains. Also, denote by X the set $\{(x_i, D_{x_i}) \mid 1 \leq i \leq \ell\}$ of pairs (variable, domain).

Definition 4.4.2 A functional denotation over $\langle F, A, X \rangle$ is defined recursively as follows:

- (i) every element of A is a functional denotation;
- (ii) every variable x_i is a functional denotation;
- (iii) if $f \in F$ is such that $f : A^n \rightarrow A$ and ξ_1, \dots, ξ_n are functional denotations then $f(\xi_1, \dots, \xi_n)$ is a functional denotation. □

We denote by \bar{F} the set of all functional denotations over $\langle F, V, X \rangle$, whenever V and X are understood.

4.5 Linked Forest Manipulation Systems

The systems that we are about to define permits the description of transformations on linked forests such that, when starting with an initial linked forest, we may reach a final linked forest by passing through a finite number of intermediate linked forests. These transformations are essentially carried out in a non-deterministic fashion, although it is possible to define completely deterministic rules for these transformations.

Notation: For any set A we denote by $P(A)$ the set of all finite subsets of A .

Definition 4.5.1 A Linked Forest Manipulation System (l.f.m.s.)

is an 8-tuple $S = (\Sigma, A, B, C, F, G, J, R)$ where

- 1) Σ is a (generally infinite) set called set of labels;
- 2) $A = \{(x, D_x) \mid x \in A_1, D_x \subseteq \Sigma\}$ is a finite subset of $A_1 \times P(\Sigma)$, where A_1 is a set of variables called label parameters, and for any $x \in A_1$, D_x is its domain;
- 3) $B = \{(y, D_y) \mid y \in B_1, D_y = T_0(\Sigma') \text{ for some } \Sigma' \subseteq \Sigma\}$ is a finite subset of $B_1 \times F_0^L(\Sigma)$, where B_1 is a set of variables called tree parameters, and for any $y \in B_1$, D_y is its domain;
- 4) $C = \vec{C} \cup \overleftarrow{C}$ is a finite set of variables called point parameters such that any element of \vec{C} has for domain $\vec{\Delta}$ and any element of

- \bar{C} has for domain $\bar{\Delta}$;
- 5) F is a finite set of function names, each being a mapping $\Sigma^n \rightarrow \Sigma$ for some n ; \bar{F} will denote the set of functional denotations over $\langle F, \Sigma, A_1 \rangle$;
 - 6) G is a finite set of tree function names, each being a mapping $(T_0^L(\Sigma))^n \rightarrow T_0^L(\Sigma)$ for some n ; \bar{G} will denote the set of functional denotations over $\langle G, T_0^L(\Sigma), B_1 \rangle$;
 - 7) J is a finite set of production labels, including two distinguished labels START and STOP;
 - 8) R is a finite set of rules; if we let $V = \Sigma \cup A_1 \cup B_1 \cup C \cup \bar{F} \cup \bar{G}$ then a rule r is an element of $P(J) \times (F_0^L(V) \times F_0^L(V) \cup F_p^L(V) \times F_p^L(V)) \times P(J) \times P(J)$ which written as $r = (L_1, \alpha_1, \alpha_2, L_2, L_3)$ verifies the following three conditions:
 - (i) α_1 and α_2 have the same skeleton, and a bijection b_r between α_1 and α_2 is given
 - (ii) every label parameter or tree parameter appearing in α_2 appears also in α_1
 - (iii) any node of α_1 or α_2 has at most one element of \bar{G} , and if it has one it should have no descendants;
 and $\Sigma, \Delta, A_1, B_1, C, F$ and J are mutually disjoint.

4.5.1 Informal Description of the System

A rule r , say $r = (L_1, \alpha_1, \alpha_2, L_2, L_3)$ will be used successfully on a linked forest Φ if α_1 can be "matched" by a substructure of Φ and that substructure is replaced by a substructure

"matching" α_2 ; the notion of "matching" needs of course to be made precise. If r cannot be used successfully on Φ then the use of r on Φ fails. In case it is used successfully, the next rule to be used is any rule $r' = (L'_1, \alpha'_1, \alpha'_2, L'_2, L'_3)$ such that L'_1 contains a label from L_2 . In case it fails the next rule to be used is any rule $r'' = (L''_1, \alpha''_1, \alpha''_2, L''_2, L''_3)$ such that L''_1 contains one of the labels of L_3 .

The first rule to be used in some transformations on a linked forest is one labelled with START. The transformations are halted whenever either a rule succeeds and has STOP in its success field, or fails and has stop in its failure field. The success field of a rule $r = (L_1, \alpha_1, \alpha_2, L_2, L_3)$ is L_2 , while its failure field is L_3 . Thus, to reflect the use made by a rule we shall write r as

$$L_1 : \alpha_1 \rightarrow \alpha_2 \quad \text{succ}(L_2) \text{ fail}(L_3) .$$

At this point a few remarks are needed to explain and justify the restrictions put on the rules.

In condition (i) the bijection b_r will be used as a basis for the replacement of a substructure corresponding to α_1 by a substructure corresponding to α_2 . In the examples that will follow, for any rule r , b_r will be implicitly indicated by the order in which α_1 and α_2 are written.

Condition (ii) is clearly needed to make the substitutions following the replacement of substructures meaningful.

Condition (iii) insures that the use of functional denotations over trees is consistent with its purpose, namely the removal of each

functional denotation over trees and its replacement by a tree attached at the same node.

In general the function $\&$ is included in G , and if so it is omitted in functional denotations, e.g. $u \& v \& w$ would be written as uvw . Also, other functions would not be given particular names especially when they are very simple, e.g. moving a label from one node of a specific tree to another. The following example will clarify these ideas.

Example 4.5 Suppose that the following rule is in a l.f.m.s.

$$L_1 : \begin{array}{c} \text{a} \\ \swarrow \quad \searrow \\ \text{b} \quad \odot \text{c, u\&v} \end{array} \longrightarrow \begin{array}{c} \text{a} \\ \swarrow \quad \searrow \\ \text{b} \quad \odot \text{d, f(u)\&v} \end{array} \quad \text{Succ}(L_2) \text{ Fail}(L_3)$$

where a, b and c are in Σ , while u and v are tree parameters with $D_u = D_v = T_0(\Sigma)$, and f is a function on trees defined by

$$f : \left\{ \begin{array}{c} \text{ } \\ \swarrow \quad \searrow \\ \text{x, END} \quad \text{y} \end{array} \mid x, y \in T_0(\Sigma) \right\} \rightarrow T_0(\Sigma)$$

with

$$f\left(\begin{array}{c} \text{ } \\ \swarrow \quad \searrow \\ \text{x, END} \quad \text{y} \end{array} \right) = \begin{array}{c} \text{ } \\ \text{y, EXEC} \end{array}$$

END and EXEC being in Σ .

We will write it in a simpler manner as

$$L_1 : \begin{array}{c} \text{a} \\ \swarrow \quad \searrow \\ \text{b} \quad \odot \text{c} \\ \quad \swarrow \quad \searrow \\ \quad \text{x} \quad \text{END} \quad \text{y} \end{array} \longrightarrow \begin{array}{c} \text{a} \\ \swarrow \quad \searrow \\ \text{b} \quad \odot \text{d} \\ \quad \swarrow \quad \searrow \\ \quad \text{v} \quad \text{EXEC} \quad \text{y} \end{array}$$

with x and y being tree parameters whose domains are D_x and D_y such that $D_x = D_y = T_0(\Sigma)$. The position of the tree parameters with respect to the nodes is used to differentiate them from the label parameters in a simple visual fashion. Notice also that the operator $\&$ is omitted.

4.5.3 Production Schemas

Definition 4.5.3.1 For $\beta_1, \beta_2 \in F(V)$, we say that (β_1, β_2) is a production schema induced by rule $r = (L_1, \alpha_1, \alpha_2, L_2, L_3)$, denoted $\beta_1 \prec \beta_2$, if there is a linked forest $\Phi \in F_0^L(\Sigma)$ such that:

- (i) α_1 , and therefore α_2 , has a corresponding substructure in Φ , say σ
- and (ii) if $\alpha_i = \{\tau_{i1}, \dots, \tau_{in}\}$, for $i = 1, 2$, with $b_r(\tau_{1j}) = \tau_{2j}$ for $j = 1, \dots, n$, and $\sigma = \{\sigma_1, \dots, \sigma_n\}$ then β_i , for $i = 1, 2$, is obtained from

$$\begin{array}{c} \sigma_1, \dots, \sigma_n \\ \Phi \\ \sigma_1 \oplus \tau_{11}, \dots, \sigma_n \oplus \tau_{1n} \end{array}$$

by removing the pivotal node label p , if any.

□

Note that all label and tree parameters appearing in β_i are the same as those appearing in α_i . Also, because those appearing in α_2 must appear in α_1 (by the definition of a l.f.m.s.), it follows that every label or tree parameter of β_2 appears in β_1 .

Now, suppose that $\{x_i\}_{i=1,\dots,\ell}$ are the label parameters of β_1 , that $\{y_j\}_{j=1,\dots,m}$ are the tree parameters of β_1 , and that $\{\vec{z}_k\}_{k=1,\dots,n} \cup \{\overleftarrow{z}_k\}_{k=1,\dots,n}$ include all the pointer parameters of β_1 and β_2 .

Let R_{β_1, β_2}^* denote the set

$$\left\{ (\gamma_1, \gamma_2) \mid \gamma_s = \beta_s \left(\begin{array}{l} x_i \leftarrow a_i \\ y_j \leftarrow b_j \\ \vec{z}_k \leftarrow \vec{c}_k, \overleftarrow{z}_k \leftarrow \overleftarrow{c}_k \end{array} \mid \begin{array}{l} 1 \leq i \leq \ell, a_i \in D_{x_i} \\ 1 \leq j \leq m, b_j \in D_{y_j} \\ 1 \leq k \leq n, c_k \in N \end{array} \right) \right\}$$

Clearly, if $(\gamma_1, \gamma_2) \in R_{\beta_1, \beta_2}^*$ then γ_1 and γ_2 may have functional denotations formed of symbols from Σ and F , or $T_0(\Sigma)$ and G .

These functional denotations do have values in Σ or $T_0(\Sigma)$ which can be obtained from the definition of the functions in F and G . Let R_{β_1, β_2} denote the result of replacing all functional denotations appearing in elements of R_{β_1, β_2}^* by their respective values. This replacement is trivial in the case of \bar{F} , and, as any node at which appears an element of \bar{G} has no descendants, the replacement of $u \in \bar{G}$ by its value v at such a node consists in the replacement in some tree expression of the set of labels at that node, say L , by $L - \{u\}[v]$.

Definition 4.5.3.2 For $\gamma_1, \gamma_2 \in F_0^L(\Sigma)$ we say that γ_2 derives from γ_1 by rule r , written $\gamma_1 \xRightarrow{r} \gamma_2$, if $(\gamma_1, \gamma_2) \in R_{\beta_1, \beta_2}$ for some β_1, β_2 such that $\beta_1 \xrightarrow{r} \beta_2$.

□

4.5.4 Transformation on Linked Forests

First, we define the transformation on a linked forest corresponding to the application of a rule of a l.f.m.s.

Definition 4.5.4.1 The binary relation \vdash is defined on $J \times F_0^L(\Sigma)$ as follows:

- (i) $(\ell_1, \gamma_1) \vdash (\ell_2, \gamma_2)$ if there is a rule $r \in R$, say $r \equiv L_1 : \alpha_1 \rightarrow \alpha_2 \text{ Succ}(L_2) \text{ Fail}(L_3)$, such that $\gamma_1 \xRightarrow{r} \gamma_2$, $\ell_1 \in L_1$ and $\ell_2 \in L_2$.
- (ii) $(\ell_1, \gamma_1) \vdash (\ell_2, \gamma_2)$ if there is a rule $r \in R$, say $r \equiv L_1 : \alpha_1 \rightarrow \alpha_2 \text{ Succ}(L_2) \text{ Fail}(L_3)$, such that for no $\gamma_2 \in F_0^L(\Sigma)$ do we have $\gamma_1 \xRightarrow{r} \gamma_2$, $\ell_1 \in L_1$ and $\ell_2 \in L_3$.

□

Notice that (i) corresponds to the successful application of r while (ii) corresponds to its failure. Let \vdash^* denote the reflexive and transitive closure of \vdash . The transformation of a linked forest into another linked forest by a linked forest manipulation system is defined next.

Definition 4.5.4.2 Given a l.f.m.s. $S = (\Sigma, A, B, C, F, G, J, R)$ and $\gamma_1, \gamma_2 \in F_0^L(\Sigma)$, we say that γ_2 is a transformation of γ_1 by system S if

$$(\text{START}, \gamma_1) \vdash^* (\text{STOP}, \gamma_2).$$

□

The following example illustrates this notion of transformation by giving a specific λ .f.m.s. and showing a transformation on a linked forest using the given system.

Note: The label ERROR is used in success and failure fields but not to label any production. It is equivalent to an empty set of labels, i.e. no production follows.

5. Model for Formal Description of Programming Languages

The model has two parts which describe formally and completely the syntax and the semantics of a programming language. The formal syntax part of the model gives the concrete and abstract syntax of program. The concrete syntax is described by context-free productions, while the abstract syntax is given by abstract syntax tree denotations. The non-context free restrictions of the language are described by means of linked forest manipulation systems.

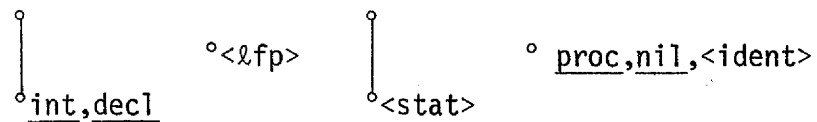
In this section we will denote by Σ_* the set of linked trees over alphabet Σ (i.e. $\Sigma_* = T_0(\Sigma)$), and by Σ_+ the set of linked trees with at least two nodes over alphabet Σ .

Definition 5.1 Let V be a set of labels, which is partitioned into two sets Σ and N , respectively, called set of terminal labels and set of non-terminal labels. An abstract syntax tree denotation over V is defined recursively as follows.

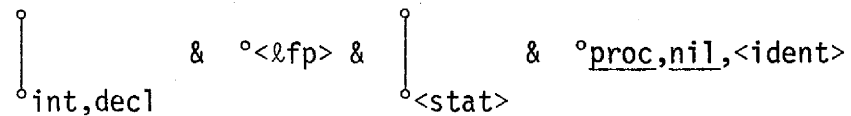
- (i) Any tree t over V in which every node has at most one non-terminal label, and if it does have one then the node has no descendant, is an abstract syntax tree denotation;
- (ii) If s and v are abstract syntax tree denotations then $s \& v$ is an abstract syntax tree denotation.

□

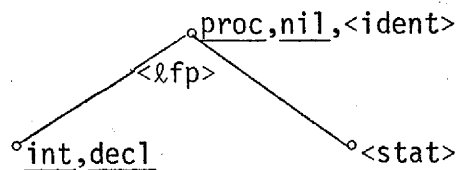
Example 5.1 Let proc, nil, int and decl be in Σ and <ident>, <lfp>, <stat> be in N . Each of the trees



is an abstract syntax tree denotation, as well as



In practice we will write this last one as



which is only an abbreviation of that abstract syntax tree denotation.

For any abstract syntax tree denotation t with non-terminal labels A_1, A_2, \dots, A_n , the linked tree resulting from t by substituting trees u_1, u_2, \dots, u_n for A_1, A_2, \dots, A_n and performing the $\&$ operation is denoted

$$\left[\begin{array}{c} A_1, \dots, A_n \\ t_{u_1, \dots, u_n} \end{array} \right] .$$

Definition 5.2

A syntax description over (Σ, F, G) for some set Σ , function names F from Σ^n to Σ , and function names G from $\Sigma_\star^n \rightarrow \Sigma_\star$, is a triplet (N, R, σ) where

N is a finite set of non-terminals

σ is a distinguished element of N called starting symbol

R is a finite set of syntax rules, each rule being a triplet (p, t, W) with

p being a context-free production, say $A \rightarrow x$ and $x \in (\Sigma \cup N)^+$ but no two non-terminals in x are the same,

t being an abstract syntax tree denotation over $\Sigma \cup N$

W being an l.f.m.s. $(\Sigma, A, B, C, F, G, J, Q)$.

□

Example 5.2 Table I gives the syntax description of programming language ALG , in which we denote by

ID a set of identifiers

INT the set of integers plus nil

SYMB = $\{+, -, *, /, \theta, :=, =\}$

KEYW = $\{\text{proc}, \text{int}, \text{decl}, \dots\}$

CNTR = $\{\text{EXEC}, \text{END}, \text{VAL}\}$.

It is a syntax description (N, R, σ) over (Σ, F, G) where

$\Sigma = \text{ID} \cup \text{INT} \cup \text{SYMB} \cup \text{KEYW} \cup \text{CNTR}$

$F = \{\text{sum}, \text{div}, \text{mult}, \dots\}$

$G = \{\oplus, \&, \dots\}$

$N = \{\langle \text{stat} \rangle, \langle \text{block} \rangle, \langle \text{var} \rangle, \dots\}$

R is given in Table I

$\sigma = \langle \text{prog} \rangle$

Any l.f.m.s. W in a syntax rule can be written as

$$W = (\Sigma, A, B, C, F, G, J, Q)$$

where $A = \{(\alpha, ID \cup KEYW), (\beta, ID \cup KEYW), (\xi, ID), (\chi, INT)\}$

$B = \{(u, \Sigma_*), (v, \Sigma_*), (w, \Sigma_*), (z, \Sigma_*)\}$

$J = \{START, STOP, ERROR, L1, \dots\}$

Q is the set of production schemas for the l.f.m.s.

Note that although in a syntax rule the context-free production, say $A \rightarrow x$, should be such that x has no two non-terminals which are the same, some syntax rules for ALG violate this condition. This is done for practical reasons, however, it should not lead to any ambiguity. For example rule 17 is

$\langle \text{exp} \rangle \rightarrow \underline{\text{if}} \langle \text{rel} \rangle \underline{\text{then}} \langle \text{exp} \rangle \underline{\text{else}} \langle \text{exp} \rangle$	<pre> graph TD if[if] --- rel["<rel>"] if --- exp1["<exp>"] if --- exp2["<exp>"] </pre>
---	---

It should be interpreted, from the formal point of view, as a shorthand for the following three rules

$\langle \text{exp} \rangle \rightarrow \underline{\text{if}} \langle \text{rel} \rangle \underline{\text{then}} \langle \text{exp}_1 \rangle \underline{\text{else}} \langle \text{exp}_2 \rangle$	<pre> graph TD if[if] --- rel["<rel>"] if --- exp1["<exp1>"] if --- exp2["<exp2>"] </pre>
$\langle \text{exp}_1 \rangle \rightarrow \langle \text{exp} \rangle$	$\circ \langle \text{exp} \rangle$
$\langle \text{exp}_2 \rangle \rightarrow \langle \text{exp} \rangle$	$\circ \langle \text{exp} \rangle$

That is the association of non-terminals, which are the same, between the right hand side of production and the corresponding abstract

syntax tree denotation, is done from left to right.

Definition 5.3 A semantics description over (Σ, F, G) , for some set Σ , function names F from Σ^n to Σ , and function names G from $\Sigma_\star^n \rightarrow \Sigma$, is a linked forest manipulation system $(\Sigma, A, B, C, F, G, J, R)$.

□

Example 5.3 Table II gives the semantics description of ALG. It is a l.f.m.s. $(\Sigma, A, B, C, F, G, J, R)$ where Σ , F and G are the same as for the syntax description;

$$A = \{(x, \text{INT}), (y, \text{INT}), (v, \{+, -, *, /, =\})\};$$

$$B = \{(t, \Sigma_\star), (u, \Sigma_\star), (v, \Sigma_\star), (w, \Sigma_\star), (z, \Sigma_+)\}, \quad \Sigma_+$$

being the set of linked trees with at least two nodes;

$$J = \{\text{START}, \text{STOP},\}$$

R is given in Table II

Definition 5.4 A language description system is a 5-tuple (Σ, F, G, S_1, S_2) where

Σ is a set of labels

F is a finite set of label function names denoting partial functions from Σ^n to Σ

G is a finite set of tree function names denoting partial functions from Σ_\star^n to Σ_\star

S_1 is a syntax description over (Σ, F, G)

S_2 is semantics description over (Σ, F, G) .

□

Example 5.4 The 5-tuple (Σ, F, G, S_1, S_2) , with Σ , F and G as defined in Example 5.2, S_1 being the syntax description of ALG, and S_2 the semantics description of ALG, is a language description system for the programming language ALG.

Given a syntax description (N, R, σ) over (Σ, F, G) , a configuration is a pair $(x, u) \in (\Sigma \cup N)^* \times \Sigma_*$, where x is called a sentential form and u an abstract tree. With every non-terminal $A \in N$ is associated a set of configurations $G(A)$ defined recursively as follows.

Definition 5.5 Configuration (α, u) is in the set of configurations associated with $A \in N$, denoted $G(A)$, if and only if there exists $r \in R$, $r = (p, t, W)$ with $p = A \rightarrow x_0 B_1 x_1 B_2 \dots B_n x_n$, such that

$$\begin{aligned} & \text{(i)} \quad \alpha = x_0 \alpha_1 x_1 \alpha_2 \dots \alpha_n x_n \\ & \text{(ii)} \quad (\alpha_i, u_i) \in G(B_i) \text{ for } i = 1, \dots, n \\ & \text{and} \quad \text{(iii)} \quad u = \begin{cases} \begin{bmatrix} B_1 \dots B_n \\ t_{u_1} \dots u_n \end{bmatrix} & \text{if } W \text{ is empty} \\ \text{a transformation of } \begin{bmatrix} B_1 \dots B_n \\ t_{u_1} \dots u_n \end{bmatrix} & \text{otherwise} \end{cases} \end{aligned}$$

□

For any syntax description $S_1 = (N, R, \sigma)$ over (Σ, F, G) , the language generated by S_1 is defined as

$$L(S_1) = \{x \in \Sigma^* \mid (x, u) \in G(\sigma) \text{ for some } u\},$$

and the abstract language generated by S_1 is defined as

$$A(S_1) = \{u \in \Sigma_* \mid (x, u) \in G(\sigma) \text{ for some } x\} .$$

Thus, if S_1 is the syntax description of a programming language the $L(S_1)$ is the set of concrete programs and $A(S_1)$ is the set of abstract programs. The translation from concrete to abstract programs is defined by

$$T_{\text{syn}}(S) = \{(x, u) \in \Sigma^* \times \Sigma_* \mid (x, u) \in G(\sigma)\} .$$

Given a semantics description $S_2 = (\Sigma, A, B, C, F, G, J, R)$, according to Definition 4.5.4.2, S_2 induces a transformation on $F_0^L(\Sigma)$ defined by

$$T_{\text{sem}}(S_2) = \{(u, v) \mid (\text{START}, v) \vdash^* (\text{STOP}, u)\} .$$

Given a language description system $S = (\Sigma, F, G, S_1, S_2)$, S induces a transformation or relation

$$T(S) = \{(x, v) \in \Sigma^* \times F_0^L(\Sigma) \mid \exists u \in \Sigma_* , \\ (x, u) \in T(S_1), (u, v) \in T(S_2)\} .$$

Intuitively, the domain of $T(s)$ is the set of programs which are syntactically correct and which execute with a normal termination. This domain is a subset of $L(S_1)$, and in general a proper subset of $L(S_1)$. A program in $L(S_1)$ but not in $T(S)$ is syntactically correct but semantically incorrect.

6. Examples

6.1 An ALG Program

To illustrate the formal description of ALG we will show several steps in the syntax processing of the following program.

```

begin
  procedure fact(value n);
    fact := if n = 1 then 1 else n*fact (n-1);
  fact(2)
end

```

We are not concerned of what happens with the computed value fact(2) , for formal description of input-output statement, see [2].

Procedure fact has a call-by-call formal parameter N . The tree associated with <fp> , as well as <lfp> , which corresponds to value n is given in Figure 6.1.1. The tree associated with <ap> , as well as <lap> , for the actual parameter n-1 is given in Figure 6.1.2. The actual parameter 2 has for tree associated with <ap> the one given in Figure 6.1.3.

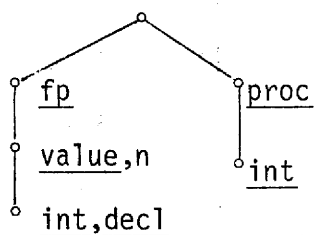


Figure 6.1.1 Tree corresponding to value n .

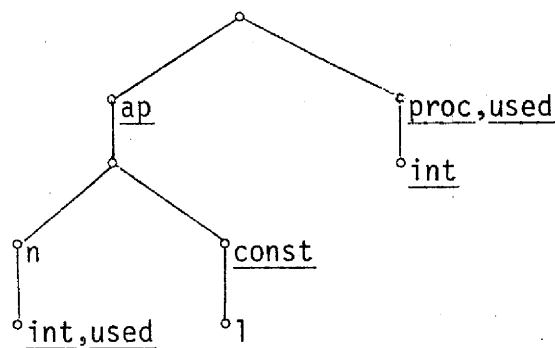


Figure 6.1.2 Tree corresponding to n-1 .

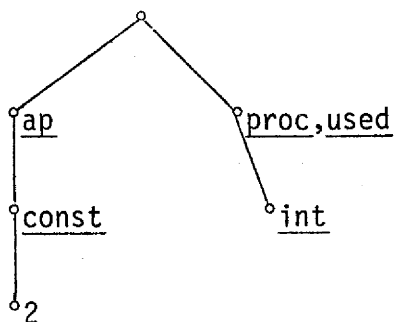


Figure 6.1.3 Tree corresponding to 2.

The tree shown in Figure 6.1.4 is the tree corresponding to the declaration of procedure fact before being affected by the transformations by the l.f.m.s. of syntax rule 43. It becomes the linked tree shown in Figure 6.1.5 after being processed by that l.f.m.s.

The statement `fact(2)` which is a call to procedure fact has for tree associated with `<stat>` the one shown in Figure 6.1.6.

Finally, the whole program has a corresponding tree which is associated with `<program>` and which is shown in Figure 6.1.7.

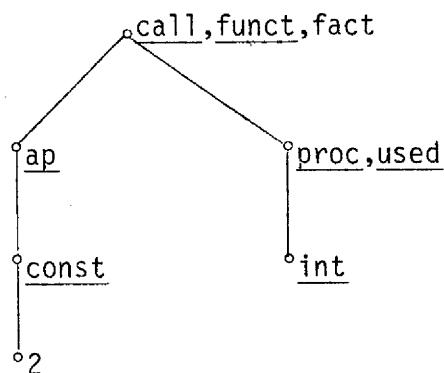


Figure 6.1.6 Tree associated with `fact(2)`.

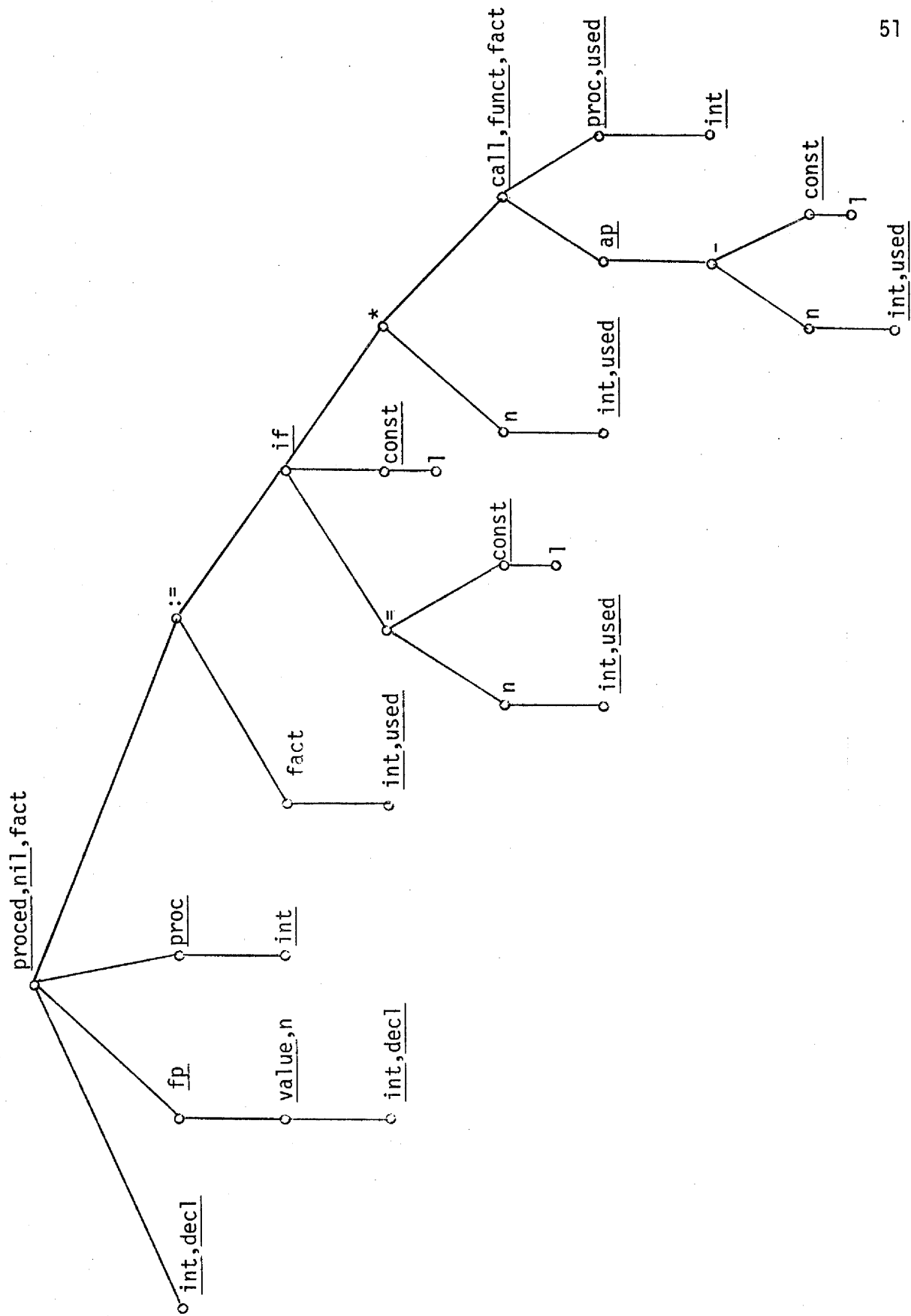


Figure 6.1.4 Tree corresponding to the declaration of procedure fact before being transformed by the l.f.m.s. of syntax rule 43.

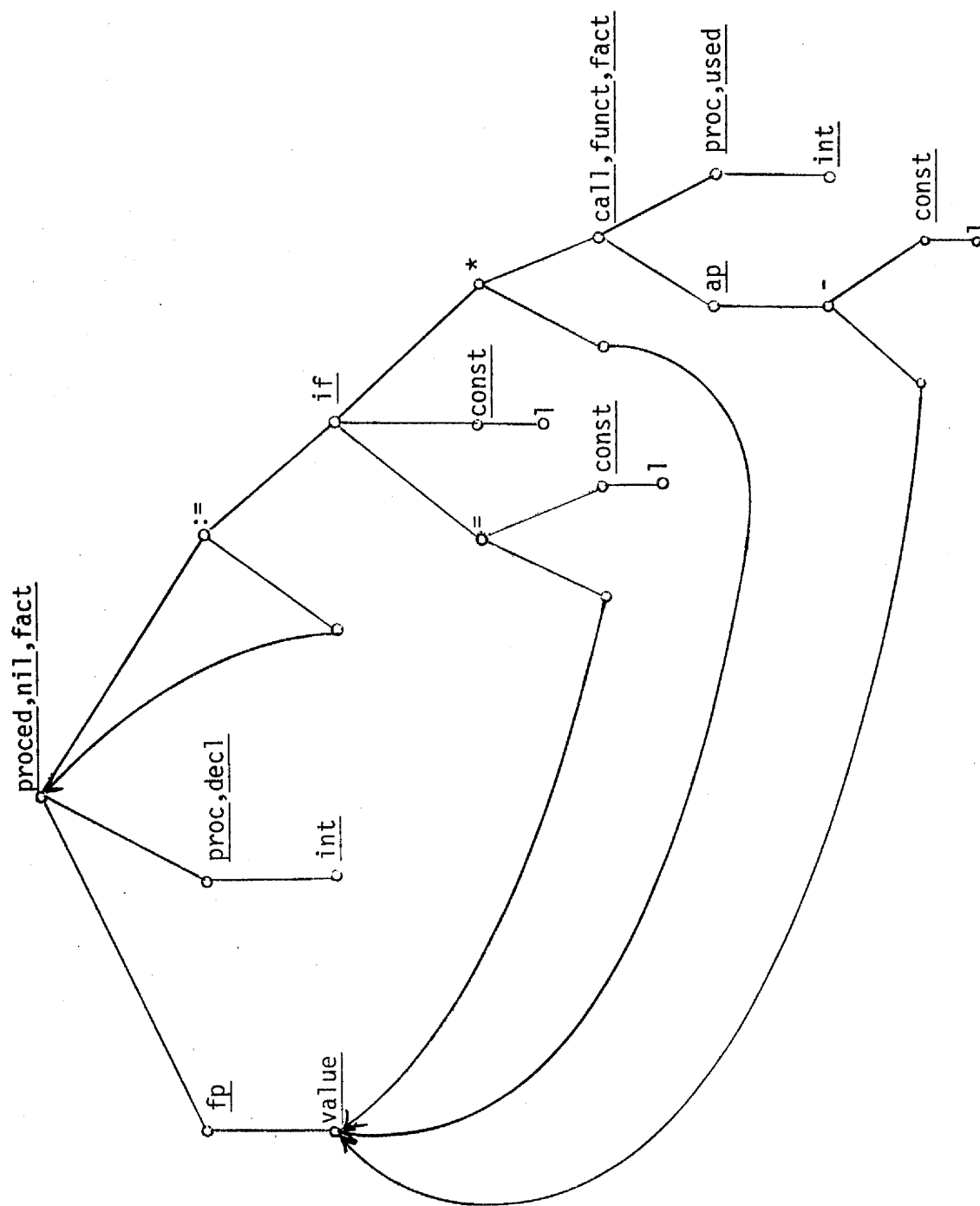


Figure 6.1.5 Tree corresponding to the declaration of procedure fact after being transformed by the l.f.m.s. of syntax rule 43.

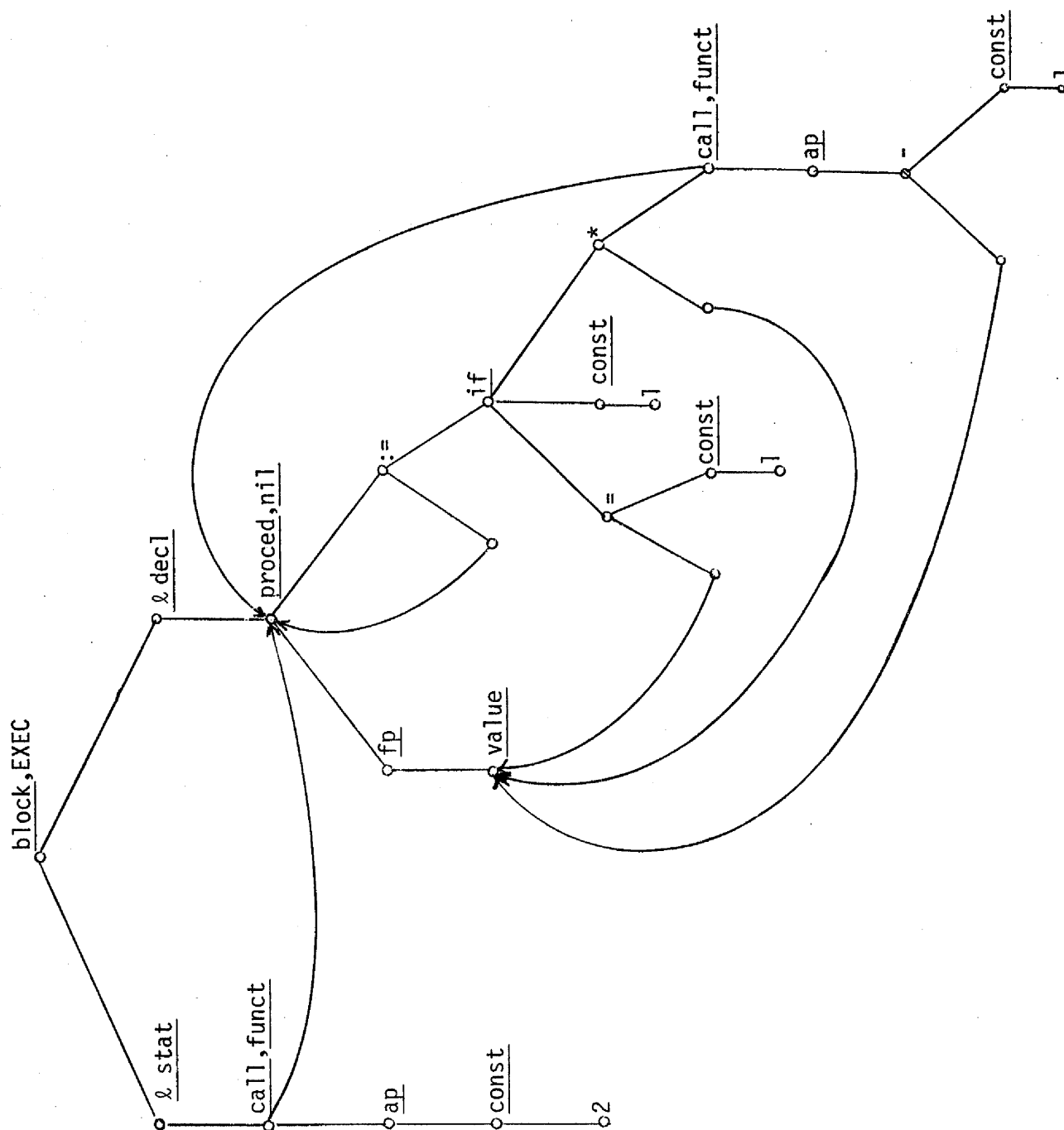


Figure 6.1.7 Tree corresponding to the whole program.

6.2 Syntax and Semantics of λ -expression

As another example of language description we give the computational semantics of λ -calculus. By computational semantics we mean the procedure of transforming any λ -expression to a reduced form. When this is done using the usual string representation of λ -expressions there is the problem of renaming of some bounded variables after each reduction. We avoid this problem completely since all the names of bounded variables are removed by the syntax description part and are replaced by pointers.

Figure 6.2.1 gives the rules of the syntax description in which $N = \{ \langle \text{ident} \rangle, \langle \text{var} \rangle, \langle \text{exp} \rangle \}$ and $\sigma = \langle \text{exp} \rangle$. The non-terminal $\langle \text{ident} \rangle$ has the same definition as in ALG.

Figure 6.2.2 gives the production schemas of the semantics descriptions for λ -expressions.

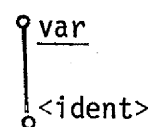

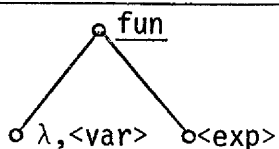


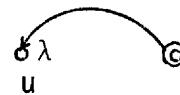




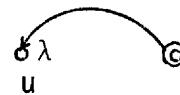




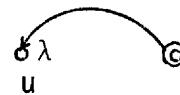


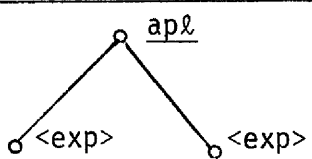
1	$\langle \text{var} \rangle \rightarrow \langle \text{ident} \rangle$															
2	$\langle \text{exp} \rangle \rightarrow \langle \text{var} \rangle$															
3	$\langle \text{exp} \rangle \rightarrow \lambda \langle \text{var} \rangle \cdot \langle \text{exp} \rangle$															
<table border="1"> <tr> <td>START</td> <td>  </td> <td>  </td> <td>→</td> <td>  </td> <td>START</td> <td>L1</td> </tr> <tr> <td>L1</td> <td>  </td> <td>→</td> <td>  </td> <td>STOP</td> <td></td> <td></td> </tr> </table>			START			→		START	L1	L1		→		STOP		
START			→		START	L1										
L1		→		STOP												
4	$\langle \text{exp} \rangle \rightarrow (\langle \text{exp} \rangle \langle \text{exp} \rangle)$															

Figure 6.2.1 Syntax description of λ -expressions

1	START		\rightarrow		L1	STOP
2	L1		\rightarrow		L1	L2
3	L2		\rightarrow		START	

Figure 6.2.2 Semantics description of λ -expressions

Table I - Syntax of ALG

1	$\langle \text{ident} \rangle \rightarrow \xi$ $\xi \in \text{ID}$	$\circ \xi$
2	$\langle \text{const} \rangle \rightarrow \chi$ $\chi \in \text{INT}$	$\circ \chi$
3	$\langle \text{var} \rangle \rightarrow \langle \text{ident} \rangle$	$\circ \langle \text{ident} \rangle$ $\circ \underline{\text{int, used}}$
4	$\langle \text{prim} \rangle \rightarrow \langle \text{var} \rangle$	$\circ \langle \text{var} \rangle$
5	$\langle \text{prim} \rangle \rightarrow \langle \text{const} \rangle$	$\circ \langle \text{const} \rangle$
6	$\langle \text{prim} \rangle \rightarrow (\langle \text{exp} \rangle)$	$\circ \langle \text{exp} \rangle$
7	$\langle \text{prim} \rangle \rightarrow \langle \text{ident} \rangle (\langle \text{lap} \rangle)$	$\circ \underline{\text{call, funct, } \langle \text{ident} \rangle}$ $\langle \text{lap} \rangle$
8	$\langle \text{factor} \rangle \rightarrow \langle \text{prim} \rangle$	$\circ \langle \text{prim} \rangle$
9	$\langle \text{factor} \rangle \rightarrow \langle \text{factor} \rangle * \langle \text{prim} \rangle$	\circ^* $\circ \langle \text{factor} \rangle$ $\circ \langle \text{prim} \rangle$
10	$\langle \text{factor} \rangle \rightarrow \langle \text{factor} \rangle / \langle \text{prim} \rangle$	$\circ /$ $\circ \langle \text{factor} \rangle$ $\circ \langle \text{prim} \rangle$
11	$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle + \langle \text{factor} \rangle$	\circ^+ $\circ \langle \text{term} \rangle$ $\circ \langle \text{factor} \rangle$

12	$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle - \langle \text{factor} \rangle$	<pre> graph TD A["-"] --- B["<term>"] A --- C["<factor>"] </pre>
13	$\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle$	<pre> graph TD A["<factor>"] </pre>
14	$\langle \text{exp} \rangle \rightarrow \langle \text{term} \rangle$	<pre> graph TD A["<term>"] </pre>
15	$\langle \text{exp} \rangle \rightarrow - \langle \text{term} \rangle$	<pre> graph TD A["-"] --- B["<term>"] </pre>
16	$\langle \text{exp} \rangle \rightarrow + \langle \text{term} \rangle$	<pre> graph TD A["<term>"] </pre>
17	$\langle \text{exp} \rangle \rightarrow \text{if } \langle \text{rel} \rangle \text{ then } \langle \text{exp} \rangle \text{ else } \langle \text{exp} \rangle$	<pre> graph TD A["if"] --- B["<rel>"] A --- C["<exp>"] A --- D["<exp>"] </pre>
18	$\langle \text{rel} \rangle \rightarrow \langle \text{exp} \rangle = \langle \text{exp} \rangle$	<pre> graph TD A["="] --- B["<exp>"] A --- C["<exp>"] </pre>
19	$\langle \text{stat} \rangle \rightarrow \langle \text{var} \rangle := \langle \text{exp} \rangle$	<pre> graph TD A[":="] --- B["<var>"] A --- C["<exp>"] </pre>
20	$\langle \text{stat} \rangle \rightarrow \text{if } \langle \text{rel} \rangle \text{ then } \langle \text{stat} \rangle \text{ else } \langle \text{stat} \rangle$	<pre> graph TD A["if"] --- B["<rel>"] A --- C["<stat>"] A --- D["<stat>"] </pre>
21	$\langle \text{stat} \rangle \rightarrow \text{call } \langle \text{ident} \rangle (\langle \text{lap} \rangle)$	<pre> graph TD A["call, proc, <ident>"] --- B["<lap>"] </pre>
22	$\langle \text{stat} \rangle \rightarrow \langle \text{ident} \rangle : \langle \text{stat} \rangle$	<pre> graph TD A["<ident>"] --- B["<stat>"] </pre>

23	$\langle \text{stat} \rangle \rightarrow \underline{\text{goto}} \langle \text{ident} \rangle$	
24	$\langle \text{stat} \rangle \rightarrow \underline{\text{begin}} \langle \text{l stat} \rangle \underline{\text{end}}$	$\circ \langle \text{l stat} \rangle$
25	$\langle \text{stat} \rangle \rightarrow \langle \text{block} \rangle$	$\circ \langle \text{block} \rangle$
26	$\langle \text{l stat} \rangle \rightarrow \langle \text{stat} \rangle$	
27	$\langle \text{l stat} \rangle \rightarrow \langle \text{l stat} \rangle ; \langle \text{stat} \rangle$	
28	$\langle \text{block} \rangle \rightarrow \underline{\text{begin}} \langle \text{l decl} \rangle ; \langle \text{l stat} \rangle \underline{\text{end}}$	

	START		ERROR	L1
	L1		L1	L2
	L2		ERROR	L3
	L3		L3	STOP

29	$\langle \text{type} \rangle \rightarrow \underline{\text{int}}$	$\circ \underline{\text{int}}$
----	--	--------------------------------

30	$\langle \text{type} \rangle \rightarrow \underline{\text{label}}$	$\circ \underline{\text{label}}$
31	$\langle \text{type} \rangle \rightarrow \underline{\text{proc}}(\langle l \text{ types} \rangle)$	$\circ \underline{\text{proc}}, \langle l \text{ types} \rangle$
32	$\langle l \text{ types} \rangle \rightarrow \langle \text{type} \rangle$	$\begin{array}{c} \circ \\ \\ \circ \langle \text{type} \rangle \end{array}$
33	$\langle l \text{ types} \rangle \rightarrow \langle l \text{ types} \rangle, \langle \text{type} \rangle$	$\begin{array}{c} \circ \langle l \text{ types} \rangle \\ \\ \circ \langle \text{type} \rangle \end{array}$
34	$\langle \text{ap} \rangle \rightarrow \langle \text{type} \rangle \langle \text{ident} \rangle$	$\begin{array}{cc} & \circ & \\ & / \quad \backslash & \\ \circ \underline{\text{ap}} & & \circ \underline{\text{proc, used}} \\ & & \\ \circ \langle \text{ident} \rangle & & \circ \langle \text{type} \rangle \end{array}$
<div> <div>START</div> <div> </div> </div> <div>→</div> <div> <div> </div> <div>STOP</div> </div>		
35	$\langle \text{ap} \rangle \rightarrow \langle \text{exp} \rangle$	$\begin{array}{cc} & \circ & \\ & / \quad \backslash & \\ \circ \underline{\text{ap}} & & \circ \\ & & \\ \circ \langle \text{exp} \rangle & & \circ \underline{\text{int}} \end{array}$
36	$\langle \text{lap} \rangle \rightarrow \langle \text{ap} \rangle$	$\circ \langle \text{ap} \rangle$
37	$\langle \text{lap} \rangle \rightarrow \langle \text{lap} \rangle, \langle \text{ap} \rangle$	$\begin{array}{cc} & \circ & \\ & / \quad \backslash & \\ \circ \langle \text{lap} \rangle & & \circ \langle \text{ap} \rangle \end{array}$

	START		STOP	
38	$\langle fp \rangle \rightarrow \underline{value} \langle ident \rangle$			
39	$\langle fp \rangle \rightarrow \langle type \rangle \langle ident \rangle$			
	START		STOP	
40	$\langle lfp \rangle \rightarrow \langle fp \rangle$	$\circ \langle fp \rangle$		
41	$\langle lfp \rangle \rightarrow \langle lfp \rangle, \langle fp \rangle$			
	START		STOP	
42	$\langle decl \rangle \rightarrow \underline{int} \langle ident \rangle$			

43

$\langle \text{decl} \rangle \rightarrow \underline{\text{procedure}} \langle \text{ident} \rangle (\langle \text{lfp} \rangle ; \langle \text{stat} \rangle)$			
START		ERROR	L1
L1		L1	L2
L2		ERROR	L3
L3		L4	
L4		L4	L5
L5		STOP	

4	$\langle \text{l decl} \rangle \rightarrow \langle \text{decl} \rangle$	
5	$\langle \text{l decl} \rangle \rightarrow \langle \text{l decl} \rangle ; \langle \text{decl} \rangle$	
6	$\langle \text{program} \rangle \rightarrow \langle \text{block} \rangle$	

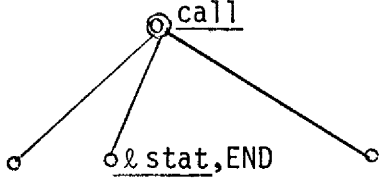
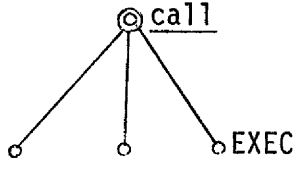
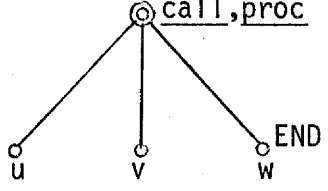
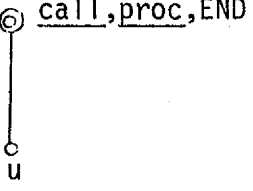
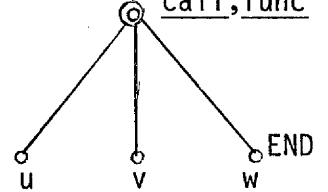
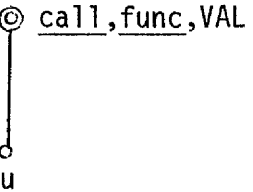
START		ERROR	STOP
-------	--	-------	------

Table II - Semantics of ALG

1	START		→			
2			→		STOP	
3			→			
4			→			
5			→			
6			→			
7			→			
8			→			
9			→			
		$v \in \{+, -, *, /, =\}$				

10				
11				
12				
13				
14				
15				
16				
17				
18				
19				

20			
21			
22			
23			
24			
25			
26			
27			
28			

29				
30				
31				

References

- [1] de Bakker, J.W. "Formal Definition of Programming Languages", Mathematical Center Tracts, Vol. 16, Mathematisch Centrum, Amsterdam (1967).
- [2] Culik II, K. "A Model for the Formal Definition of Programming Languages", Intern. J. Computer Maths., Section A, Vol. 3, pp.315-345.
- [3] Farah, M. "A Formal Description of ALTRAN Using Linked Forest Manipulation Systems", Res. Report CS-73-08, Department of Computer Science, University of Waterloo.
- [4] Farah, M. "Correctness of a LUCID Interpreter Based on Linked Forest Manipulation Systems", Res. Report CS-77-07, Department of Computer Science, University of Waterloo.
- [5] di Forino, C. Generalized Markov Algorithms and Automata in "Automata Theory", edited by E.R. Caianiello, 115-130, Academic press, New York (1966).
- [6] Knuth, D.E. "Semantics of Context-Free Languages", Math. System Theory, Vol. 2, pp. 127-145.
- [7] Lucas, P., Lauer, P., and Stigleitner, H. "Method and Notation for the Formal Definition of Programming Languages", Technical Report TR 25.087, IBM Laboratory, Vienna.
- [8] McCarthy, J. "A Basis for a Mathematical Theory of Computation", Computer Programming and Formal Methods, pp. 33-69, North-Holland (1963).
- [9] Rosen, B.K. "Tree-Manipulating Systems and Church-Rosse Theorem", J. ACM, Vol. 20, No. 1, January 1973, pp. 160-187.
- [10] Rosenkrantz, D.J. "Programmed Grammars and Classes of Formal Languages", J. ACM, Vol. 16, No. 1, January 1969, pp. 107-131.
- [11] Wegner, P. "The Vienna Definition Language", ACM Computing Survey, Vol. 4, No. 1, March 1972.
- [12] Wirth, N. and Weber, H. EULER, A Generalization of ALGOL, and its Formal Definition, Comm. ACM, 9, 13-23, 89-99 (1966).
- [13] Zoltan, A.C. "A Formal Defintion of ALGOL 60 Using Linked Forest Manipulation Systems", Research Report CSRR-1072, Department of Computer Science, University of Waterloo.