TOWARDS BETTER STRUCTURED DEFINITIONS OF PROGRAMMING LANGUAGES

by

R. Kurki-Suonio

STAN-CS-75-500
SEPTEMBER 1975

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

Towards Better Structured Definitions of Programming Languages

Reino Kurki-Suonio

Abstract

The use of abstract syntax and a behavioral model is discussed from the view-point of structuring the complexity in definitions of programming languages. A formalism for abstract syntax is presented which reflects the possibility of having one defining occurrence and an arbitrary number of applied occurrences of objects. Attributes can be associated with such a syntax for restricting the set of objects generated, and for defining character string representations and semantic interpretations for the objects. A system of co-operating automata, described by another abstract syntax, is proposed as a behavioral model for semantic definition.

This research was supported in part by National Science Foundation grant DCR72-03752 A02, IBM Corporation, and by the Academy of Finland. Reproduction in whole or in part is permitted for any purpose of the United States Government.

1. Introduction.

In a behavioral definition of a programming language, the meaning of a program is defined in terms of its dynamic execution or interpretation. In other words, a behavioral definition gives an abstraction of the run-time behavior of programs, not only of the input-output mappings performed by them. To be useful, the abstractions introduced by the definition should provide appropriate mental tools for intuitive understanding of program behavior. Such models and tools are needed both when implementing and when teaching programming languages.

Rather than defining directly the meaning of a program given as a character string, it is useful to introduce an intermediate level of abstract programs. The meaning of an abstract program can be defined by postulating a machine to execute abstract programs, or by giving an interpreting automaton for them, or by providing an abstract compiler which translates abstract programs into some kinds of automata. The third alternative introduces to the definition another level which will be called a behavioral model. This leads to the following levels of definition:

- string representations,
- abstract programs,
- behavioral model.

The monumental definitions of PL/I [25] and Algol 68 [24] can both be characterized as behavioral definitions. In the former, abstract PL/I programs are essentially syntax trees of string programs, with some of the complexities of string representations removed, and an interpreter is provided to determine their meanings. In the latter, careful language design and a powerful syntax formalism make it convenient to use syntax trees themselves as abstract programs, and a machine, or "elaboration" mechanism, is given for their execution. In addition, [24] has an additional "surface level" in which actual representations are provided for the "abstract" character set used at the other levels.

Levels of language description are intended to provide a natural classification of language properties according to their "deepness". It is assumed in the following that abstract programs are free from features not essential to their structure and meaning. Those properties

which are associated with the mapping between string representations and abstract programs can then be called surface properties, while other properties are deep properties. Concerning the latter, the behavioral model is a natural level for the most fundamental concepts underlying the language. Ideally, these concepts should be "universals", common to a larger family of languages.

No language definition seems to have made full use of such structuring of language concepts by levels of language description. For instance, abstract PL/I programs indicate such surface properties as the number of redundant parentheses and the particular choices for identifiers. In Algol68 Report, on the other hand, the powerful syntax formalism allows extensive use of syntactic definition. For this reason there has not been need for clear separation between surface properties and deep properties. It appears to the author that the lack of such separation is a greater difficulty for an uninitiated reader of [24] than the formalism of w-grammars. The possibility of including the most fundamental concepts in a behavioral model has not been utilized in any of these two definitions, as abstract programs are interpreted or executed directly.

As mentioned above, a behavioral model should reflect some of the most fundamental concepts in programming. If the model is based on "typical" properties of present computers, as the implicit models in the definitions of Fortran and Cobol, it does not give any mental tools for understanding programming and programming languages. Even when higher-level models are used in a definition, they might be restricted by the current technology. For instance, a stack-oriented behavioral model might be suitable for a language not requiring more general techniques for storage allocation, but its use would be limited to a restricted class of languages. Although no universally applicable behavioral model is to be expected, it appears that useful models could be given for families of languages based on the same conceptual backgrounds. An important aspect of a behavioral model is the way it models parallel processes. The emerging understanding of how to manage such processes should provide appropriate ways of incorporating them in to a model.

The thesis of this paper is that the three levels of language definition described above correspond in a natural way to the abstractions that a language designer has in mind. Submitting such a vision of the language to the reader would be very helpful in language definitions and textbooks. The techniques proposed for this purpose are based on a generalization of abstract syntax [17,18] with attributes associated with the nonterminals [11]. A simple example will be used throughout to illustrate the techniques. A system of automata, communicating with each other, is outlined as a behavioral model.

It is understood that the separation of language properties according to "deepness" is subject to similar non-objective criteria as language design. In part, this is due to programming languages being man-made, not something given to us like the physical world. In particular, any suggestion for language universals can be contradicted by designing another language specifically for this purpose. The behavioral model outlined below will be directed towards block structured languages. It is included mainly to illustrate the general ideas of the paper. For practical purposes it should probably be extended with further capabilities reflecting some other fundamental concepts in programming.

The ideas developed in this paper have been induced and influenced by a number of other papers, some of which are mentioned in the bibliography. To pick out the most important ones after Algol 60 [20], I would like to acknowledge Landin's work on h-calculus models [13,14], Strachey's efforts to single out fundamental concepts of programming languages [22,23], the definitions of PL/I [25] and Algol 68 [24], Johnston's contour model [10], the class concept of Simula 67 [5] and its effect on the development of programming-concepts [1,2,8,9,19], as well as Knuth's idea of synthesized and inherited attributes [11].

2. Formalism for Abstract Syntax.

Abstract syntax deals with objects having certain kinds of structural relationships. More precisely, an object may have other objects as its immediate components. The transitive closure of this relation will be called the component relation. The set consisting of an object x and all its components will be denoted by c(x) .

Generalizing the concept of an abstract syntax [16], we shall allow an object to be an immediate component of several objects. Also, it will be possible for an object to be its own component, although we shall introduce quite severe restrictions in these respects. Only finite structures will be considered, i.e., it will be assumed that c(x) is finite for each object x .

Abstract syntax expresses structural relationships in terms of structural productions resembling ordinary context-free productions for formal languages. Each nonterminal of an abstract syntax represents a class of objects, and these classes are assumed to be disjoint for different nonterminals. For each non-terminal there are productions indicating the different ways in which objects of this class may- have other objects as their immediate components. The productions for one class are also assumed to be disjoint in the sense that no object will consist of immediate components according to more than one production.

As an example, the production

e:Expr → SUM⟨el:Expr, e2:Expr⟩

indicates that an object of the class Expr may have two immediate components, both belonging to this same class. The label SUM is used to identify the production, and labels e , el , e2 are introduced to identify objects in the context of this production. Obviously, triple labelling -- nonterminal names, production names, and object names in productions -- could be avoided in our notations. It is felt, however, that this redundancy is a convenience when using the formalism.

In order to cope with situations when an object is an immediate component of more than one object, we distinguish between two cases of immediate component relation, called primary and secondary immediate component relations. Furthermore, we require that every object is a

primary immediate component of at most one object, and that the transitive closure of the primary immediate component relation, called primary component relation, is non-reflexive. This means that objects are connected by a tree structure, with additional connections indicating secondary components. The set consisting of an object x and all. its primary components will be denoted by p(x) .

In the productions a secondary component will be indicated by enclosing a non-terminal in parentheses. For example, the production

s:St → ASS((v:Var), e:Expr)

indicates that an object of class St may consist of a secondary component of class Var together with a primary component of class Expr. Corresponding to whether a component is primary or secondary, it is called a defining or applied occurrence of that object. Correspondingly, occurrences of nonterminals in the right-hand sides of productions will also be called defining or applied.

Corresponding to terminal symbols in a context-free grammar, an abstract syntax has terminal sets which are disjoint from the sets determined by the nonterminals. For instance, if N denotes the set of nonnegative integers, the production

e:Expr → CONST⟨(n:N)⟩

indicates that an object of class Expr may have an arbitrary nonnegative integer as its only component. Such terminals are given as applied occurrences to emphasize that they have no defining occurrence in the syntax.

Another way to terminate a structure is provided by productions indicating explicitly that an object of a non-terminal class is atomic with regard to the component relation. This is illustrated by the production

v:Var → ATOM⟨ ⟩.

To summarize, an abstract syntax consists of a finite family of terminal sets, a finite set of' non-terminals, and a finite set of productions, where each production indicates one possibility for objects of a nonterminal class to consist of primary and secondary components of

given nonterminal classes and/or terminal sets. These possibilities are assumed to be disjoint in the sense that each object has components as indicated by exactly one production. (However, several productions for the same nonterminalmay be similar, except for production identifications.) One of the nonterminals is designated as the starting non-terminal, and an object x is generated by the abstract syntax, iff

- x belongs to the class of the starting nonterminal,
- the primary component relation imposes a tree structure on $p(x)$,
- elements of $c(x)-p(x)$ are elements of terminal sets.

Our previous examples of structural productions involved a fixed number of components. For notational-convenience we introduce the notation

lab: Object*

to denote a sequence of an arbitrary finite number of components of the same class. For a sequence of n components, the individual members are identified by indexing the label used: lab[1], . . .,lab[n] . When used without indexing, the label will identify the whole set. The index set $\{1,\ldots,n\}$ of a label will be denoted by $\iota(\text{lab})$.

As an example, let us consider the following abstract syntax, where N denotes the set of nonnegative integers:

p:Prog → PROG⟨b:Block⟩
b:Block → BLOCK⟨v:Var*,s:St*⟩
s:St → COMP⟨b:Block⟩
→ ASS⟨⟨v:Var⟩,e:Expr⟩
e:Expr → CONST⟨⟨n:N⟩⟩
→ VAR⟨⟨v:Var⟩⟩
→ SUM⟨e1:Expr, e2:Expr⟩
→ PROD⟨e1:Expr, e2:Expr⟩
→ NEG⟨e1:Expr⟩
v:Var → ATOM⟨ ⟩ .

Intuitively this abstract syntax is meant to generate block structured programs with assignment statements involving simple arithmetic expressions over integers and integer variables. However, no scope rules for variables have been incorporated at this stage.

Any object generated by an abstract syntax can be represented as a directed graph with suitable labelling in an obvious way. For instance, one of the objects generated by the above syntax could be represented as the graph of Figure 1. Each node in the graph corresponds to one object. An object is marked by its nonterminal class or by a notation for an element of a terminal set. Continuous lines indicate defining occurrences, and dotted lines indicate applied occurrences. The productions involved are indicated by production labels under nonterminal names. Object identification (in the context of productions applied) is given by attaching appropriate labels to both nodes and edges.

Synthesized and inherited attributes [11] can be associated with this kind of abstract syntax in the same way as with conventional context-free grammars. However, no attribute definitions should be associated with applied occurrences of nonterminals. For instance, we can associate an attribute $\mathcal{a}$ with the above syntax as follows:

p:Prog	→	PROG⟨b:Block⟩	$\mathcal{a}_p := \mathcal{a}_b$
b:Block	→	BLOCK⟨v:Var^x, s:St^x⟩	$\mathcal{a}_b := \bigcup_{i \in x(s)} \mathcal{a}_{s[i]} - v$
s:St	→	COMP⟨b:Block⟩	$\mathcal{a}_s := \mathcal{a}_b$
	→	ASS⟨(v:Var),e:Expr⟩	$:= \{v\} \cup \mathcal{a}_e$
e:Expr	→	CONST⟨(n:N)⟩	$\mathcal{a}_e := \emptyset$
	→	VAR⟨(v:Var)⟩	$:= \{v\}$
	→	SUM⟨e1:Expr,e2:Expr⟩	$:= \mathcal{a}_{e1} \cup \mathcal{a}_{e2}$
	→	PROD⟨e1:Expr,e2:Expr⟩	$:= \mathcal{a}_{e1} \cup \mathcal{a}_{e2}$
	→	NEG⟨e1:Expr⟩	$:= \mathcal{a}_{e1}$
v:Var	→	ATOM⟨ ⟩	

For all nonterminals (except Var) the attribute $\mathcal{a}$ is set-valued and synthesized. Association with different objects in the context of a production is indicated by appropriate subscripting of $\mathcal{a}$. The intuitive meaning of $\mathcal{a}$ is the set of those variables which have been used but not properly declared in the structural unit in question. The

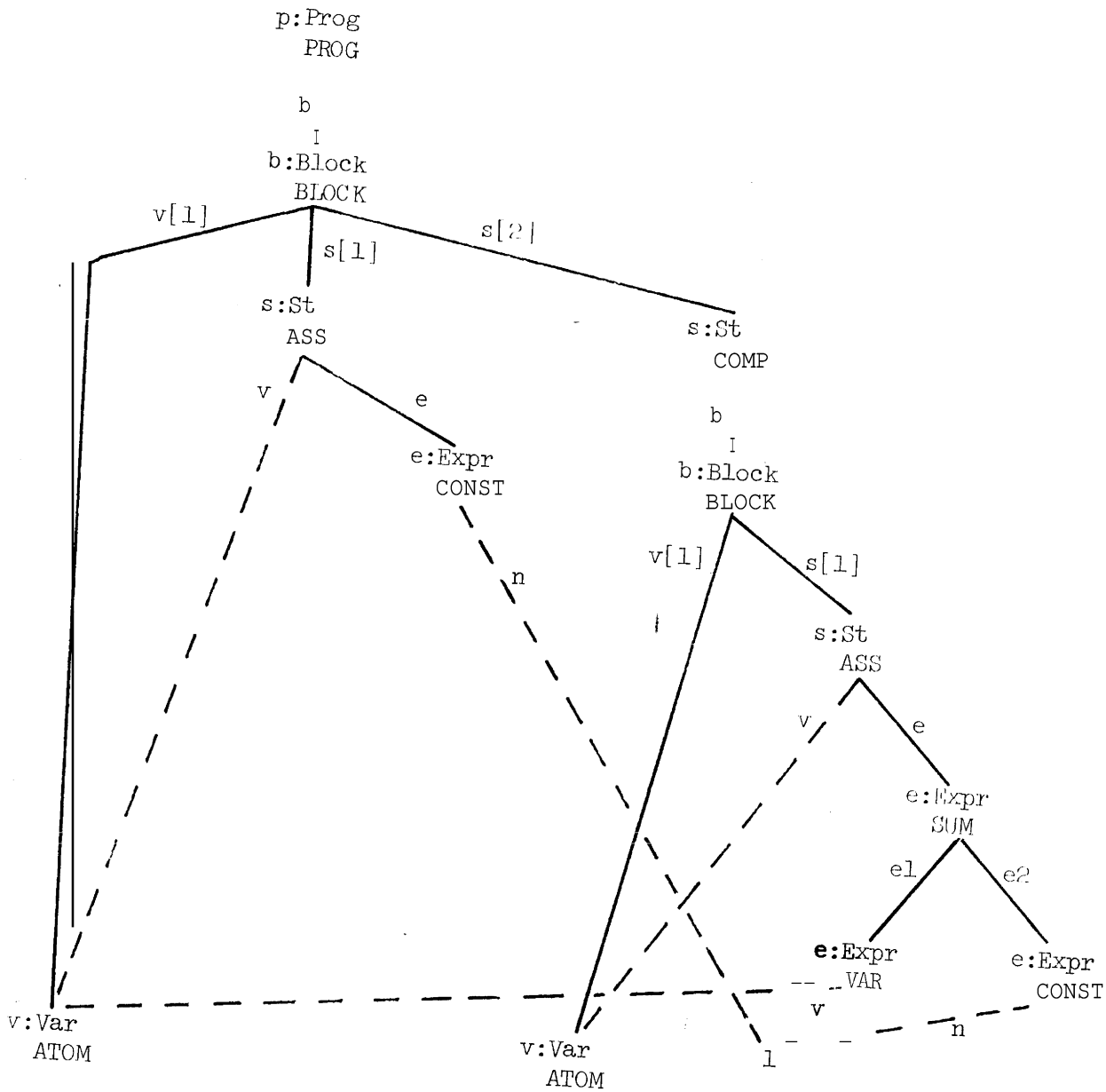
p:Prog
PROG
b
I
b:Block
BLOCK
v[1]
s[1]
s[2]
s:St
ASS
s:St
COMP
v
e
e:Expr
CONST
b
I
b:Block
BLOCK
v[1]
s[1]
n
s:St
ASS
v
e
e:Expr
SUM
e1
e2
e:Expr
VAR
e:Expr
CONST
v
n
v:Var
ATOM
v:Var
ATOM
1

Figure 1

set of objects

$$\{x \in Prog \mid a_x = \emptyset\}$$

would now consist of exactly those objects of class Prog which intuitively correspond to programs whose variables are used only when they have been declared in an enclosing block.

5. Associating String Representations with Abstract Programs.

The attribute technique is also useful for associating concrete character string representations with objects generated by an abstract syntax. Since one object may, in general, have many different representations, we shall use nondeterministic attributes. In other words, an attribute definition may provide a set of possible values, preceded by the monadic selection operator σ. By this notation we understand that any element of the set can be taken as the value of the attribute in question. When an attribute value is determined by a conditional expression, nondeterminism may also appear in the form that more than one or none of the conditions hold. If none of the conditions hold, the attribute value is undefined, and some other choices for attribute values must be changed to get out of this dead end. Vertical bars (|) will be used to separate alternatives in conditional expressions, and each condition will be separated from the corresponding expression by a colon (:).

As an example, let us assign a string-valued attribute $\mathcal{T}$ and an integer-valued attribute $\mathcal{P}$ to the previous abstract syntax, as given in Figure 2. The following clarifications should be added to the notations used:

- The notation $\mathcal{C}^{z}_{i=1,z(v)}(x_i)$ stands for the concatenation $x_1 z x_2 z \ldots z x_{|z(v)|}$ if $|z(v)| > 1$, for x_1 if $|z(v)| = 1$, and for the empty string if $|z(v)| = 0$.
- Symbols L and D stand for the sets of letters and digits, corresponding+

p:Prog →	PROG⟨b:Block⟩	$\tau_p := \tau_b$
b:Block →	BLOCK⟨v:Var*, s:St*⟩	$\tau_b := \iota(v) = \emptyset : \underline{\text{begin}}\ \mathcal{C}'_{i \in \iota(s)}(\tau_{s[i]})\ \underline{\text{end}} \mid$ $\iota(v) \neq \emptyset \wedge \forall v1, v2 \in v \cup \mathcal{A}_b\ (v1 \neq v2 \Rightarrow \tau_{v1} \neq \tau_{v2}) :$ $\underline{\text{begin}}\ \underline{\text{new}}\ \mathcal{C}'_{i \in \iota(v)}(\tau_{v[i]});\ \mathcal{C}'_{i \in \iota(s)}(\tau_{s[i]})\ \underline{\text{end}}$
s:St →	COMP⟨b:Block⟩	$\tau_s := \tau_b$
→	ASS⟨(v:Var), e:Expr⟩	$\tau_s := \tau_v \leftarrow \tau_e \qquad \rho_e := 0$
e:Expr →	CONST⟨(n:N)⟩	$\tau_e := \sigma\{(^m\ c^*\ \tau_n)^m \mid m \geq 0\}$ $\tau_n := n=0 : 0 \mid n=1 : 1 \mid n=2 : 2 \mid n=3 : 3 \mid n=4 : 4 \mid$ $n=5 : 5 \mid n=6 : 6 \mid n=7 : 7 \mid n=8 : 8 \mid n=9 : 9 \mid$ $\underline{\text{else}} : \tau_{\lfloor n/10 \rfloor}\ \tau_{n \bmod 10}$
→	VAR⟨(v:Var)⟩	$\tau_e := \sigma\{(^m\ \tau_v)^m \mid m \geq 0\}$
→	SUM⟨e1:Expr, e2:Expr⟩	$\tau_e := \rho_e = 0 : \sigma\{(^m\ \tau_{e1} + \tau_{e2})^m \mid m \geq 0\}$ $\underline{\text{else}} : \sigma\{(^m\ \tau_{e1} + \tau_{e2})^m \mid m > 0\} \quad \rho_{e1} := 0 \quad \rho_{e2} := 1$
→	PROD⟨e1:Expr, e2:Expr⟩	$\tau_e := \rho_e \leq 1 : \sigma\{(^m\ \tau_{e1} * \tau_{e2})^m \mid m \geq 0\}$ $\underline{\text{else}} : \sigma\{(^m\ \tau_{e1} * \tau_{e2})^m \mid m > 0\} \quad \rho_{e1} := 1 \quad \rho_{e2} := 2$
→	NEG⟨e1:Expr⟩	$\tau_e := \sigma\{(^m\ -\ \tau_{e1})^m \mid m \geq 0\} \qquad \rho_{e1} := 0$
v:Var →	ATOM⟨ ⟩	$\tau_v := \sigma(L(L \cup D)^*)$

Figure 2

- The expressions (sets) given for string-valued attributes are to be understood as strings (sets of strings). No notational distinction has been made between string expressions and arithmetic expressions.

- Attributes can also be associated with element=; of terminal sets. In this example, attribute τ is defined for objects $n \in N$, and its definition involves making use of arithmetic operations defined in N .

For any $x \in Prog$, the possible values of τ_x are now defined as the possible string representations of x . Nondeterminism in attribute definitions allows for arbitrary redundancy in parenthesizing, and for arbitrary variable names . The definition of τ_{Block} guarantees that the same name cannot be used for two different variables in the same block.

In this particular case it can be easily verified that τ_x has values for all objects $x \in Prog$. In general, this would not need to be the case, because of the "dead ends" involved in attribute definitions. One possible value of τ_x for the particular x given in Figure 1 is

```
begin new a;
      a ← 1;
      begin new b;
            b ← a+1
      end
end
```

4. Behavioral Models.

As mentioned previously, one would wish a behavioral model to reflect fundamental concepts of a language which would be common for a larger group of languages. Typically, such deep properties would be related to concepts like block structure, data types, and sequential and parallel actions .

Models based on h-calculus [13,14] and the more visual contour model [10] mainly reflect block structuring, and it would be very difficult to understand block structured languages without these or similar mental models. However, data types and parallelism seem to require different kinds of models.

The main problems with data structures are associated with sharing through explicit or implicit pointers, and with control over access and updating. Sharing can be expressed in terms of relations which can be visualized as directed graphs, as e.g. in [4,5,12].

As an example, let us consider some properties of data structures in Algol68. For simplicity, routines will be omitted from the discussion. The relations used in the behavioral model are the following: a "name" "refers to" a value, and a "structured" or "multiple" object has components together with selectors. Superficial knowledge of these relations and analogy with other languages with similar structures might lead to the incorrect visualization given in Figure 3. Different data objects are grouped there into "plain values" V , non-structured names N , and into "structured" and "multiple" objects S . Arrows starting from N indicate the relation "to refer to" for non-structured names, and those starting from S indicate component relations. All "refer to" chains and component chains are assumed to be finite. Objects in V are assumed to exist independently of any program; objects in N and S are "created" during program execution. The relation "refer to" is changed by assignments; assignment to an element of s is understood as simultaneous assignment to all of its components.

This visualization is shown insufficient by the existence of structured objects to which no assignment can be made, even if the components can be updated individually. This leads to the modification given in Figure 4. Non-updatable structured objects are grouped into S' elements of S are restricted to have no components in V or S' ; elements of N are restricted not to refer to any object in S' ; each element x of s is postulated to refer to an object $x' \in S'$ such that the components of x' are the values referred to by the components of x Objects x and y in Figure 4 are otherwise similar, but y can be updated only by individual assignments to its components.

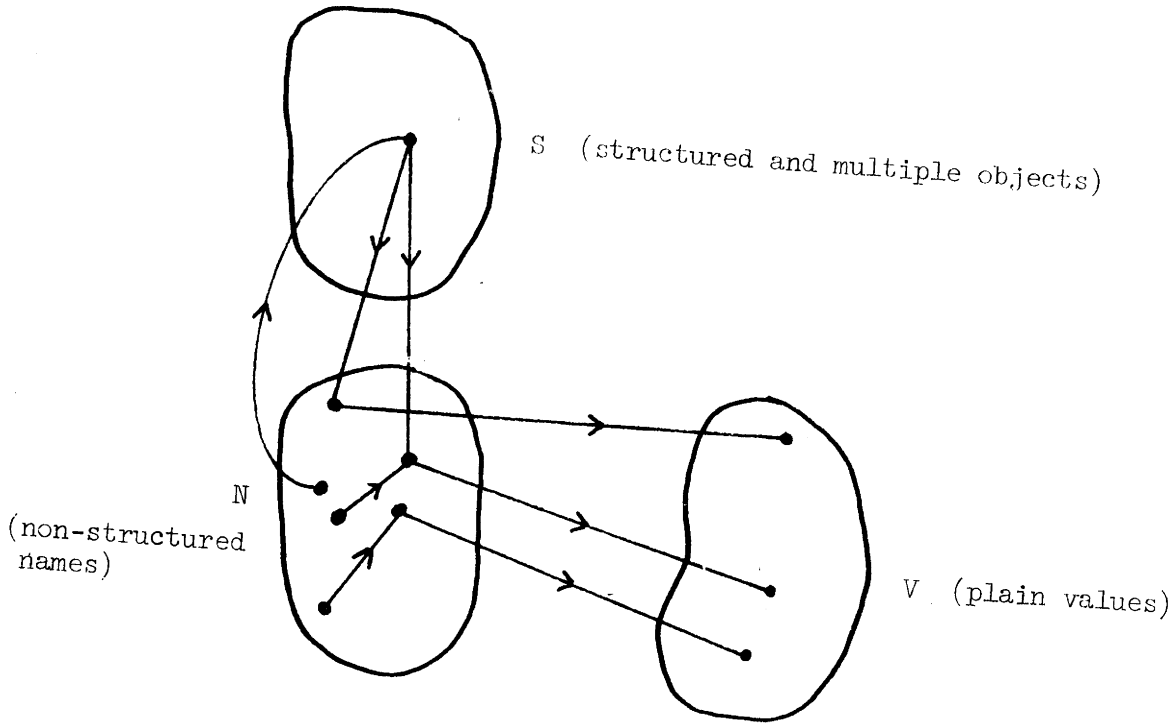

Figure 3

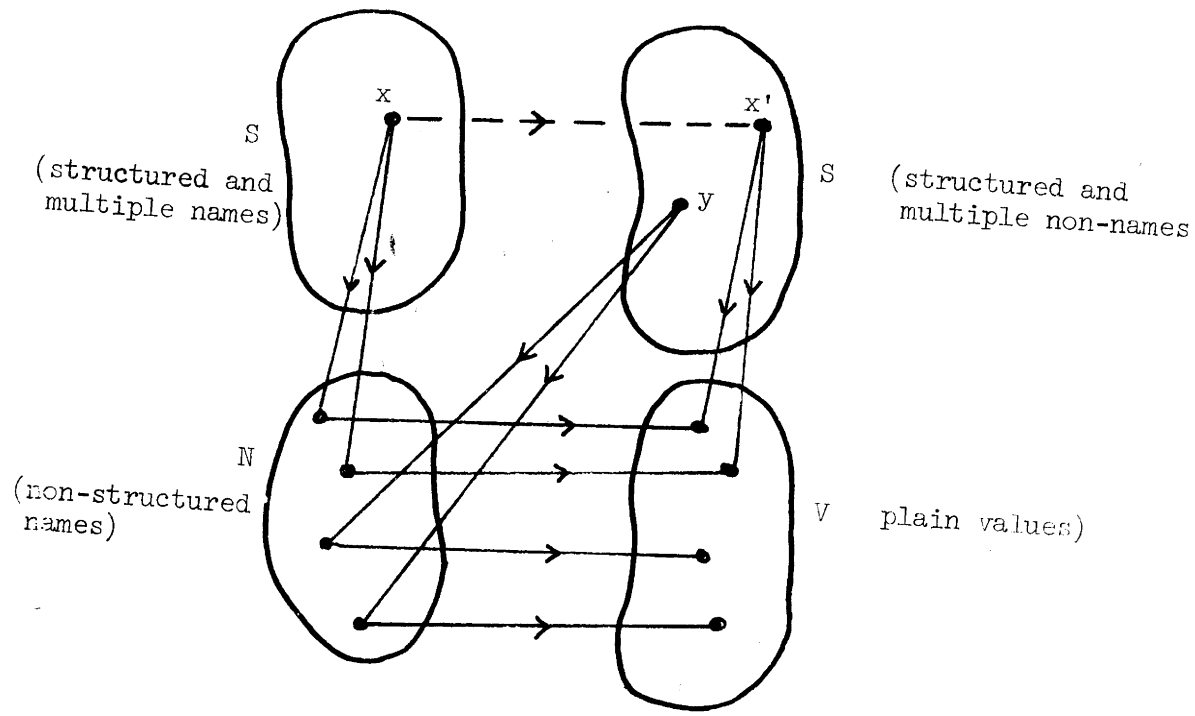

Figure 4

Examples of this kind show the usefulness of graphs as visual aids in understanding data structures, especially when dealing with several languages where similar structures are understood differently. Such a model is suitable for expressing properties of sharing, but it is insufficient for indicating aspects of protection on data access and updating, except for very limited special cases. Also, it tends to treat data objects as passive objects, and this causes difficulties in dealing with routines or procedures as data.

One way to overcome the problems mentioned above is to treat data objects as active objects, as suggested by the class concept in Simula 67 [3]. This leads to the view that each data object is some kind of an automaton, which is initialized when created, and which is thereafter able to communicate with other automata by responding to certain requests. Access to the "value" of an object or to its "components", and any updating, would only be possible through such requests sent to the data object in question. Operations defined for data types could also be evoked by such requests only. This kind of a behavioral model also provides a natural framework for parallelism, as all data objects would be envisaged as automata "living" simultaneously. Notice that such a model deviates in several respects from Simula 67, where only quasiparallelism is allowed, unrestricted external access to "attributes" of' data objects is permitted, and primitive data items are not treated as independent active objects.

Primitive values like integers, reals, Booleans, etc., can now be considered as primitive automata, existing independently of any program. In other words, no such automata can be created or initialized by a program. The only way such automata can communicate with other automata is to respond to certain requests. Parameters associated with such requests, and the responses, are always automata themselves. For instance, if the automaton for an integer i is asked for its negation, the response gives the automaton for -i . Similarly, if an automaton for a-truth value is sent a request corresponding to if-then-else -operation, together with two automata as parameter:;, it return:: one of the parameters as the response.

Variables are now thought of as automata which can be "created" by a program. In other words, among the objects "existing" outside a program, there are "schemes" which can be used to create "new" automata. Using such a scheme results in the creation and initialization of an automaton, which can thereafter respond to certain requests. For instance, there are primitive schemes for integer variables. After initialization, an automaton created by such a scheme is capable of responding to requests about its "value", and to update this value. Structured objects and arrays have somewhat more complicated schemes, which must also be primitives in the behavioral model.

A block structured program can now be viewed as a hierarchical structure of schemes of automata, such that each new automaton has certain schemes available for the creation of further automata. Program execution is then modelled as the creation of a system of automata communicating with each other and creating new automata. All the "real" work of the program is done by the primitive automata existing -independently of the program.

Parallel operation of a system of automata is controlled by allowing each automaton to process only one request at one time. Further requests are thought of as waiting for their turns. Since a response is expected for each request, the requesting automaton has to wait for the request to be processed. Parallel processes can be created by allowing the initialization of an automaton to proceed simultaneously with continuing the process which evoked this initialization.

Comparing with the definition of Algol 68, it is pointed out that its behavioral definition is essentially based on monoprogramming, as "collateral elaboration" is defined by "merging" the sequences of primitive actions corresponding to the constituents. When the constituents involve updating the same data objects, that model doe:: not determine what happens, as the primitive actions are not specified. In the model proposed here, all operations on the same data item always exclude each other in time. Since integers, for instance, are considered as individual automata, we get the curious restriction of parallelism, however, that two operations involving the same numbers cannot be performed simultaneously. This situation could be avoided by letting

different occurrences of the same number correspond to distinct but similar automata.

This kind of a model for programs is compatible with many of the current views of operating systems and data bases, and with the principles of structured programming. In fact, similar extensions of the ideas in Simula 67 have been applied to control parallelism in the programming of operating systems [1,2, 9], and in the design of languages for structured programming [6,15].

An operating system can be characterized as a collection of processes communicating with each other. Although these processes are usually executed on a single processor, this view becomes especially clear when thinking of each process as having a processor of its own. Such a view has advantages even when used only as a mental model to understand a conventional operating system, since it seems easier to introduce restrictions on parallelism into an inherently parallel model, than to introduce suitable quasiparallelism into a basically sequential model. However, it also appears to be quite feasible to construct multiprocessors where different kinds of operating system duties have been distributed to different processors. In particular, it seems promising to use dedicated processors for data bases. In such a case parallelism of the model would correspond to real parallelism in the operating system. Note, however, that aspects of protection in operating systems would need further elaboration of the model.

In addition to parallelism, this model reflects the principle of information hiding [21] in a natural way. Besides being an important design principle, information hiding is needed to describe data independence in programs using data bases. In this case it is not only a property of the model that direct access and updating of passive data items is prevented. On the contrary, it is then an essential feature of the system to be modelled, that all external access must take place through explicit requests. For instance, a program cannot "know" whether a piece of data, which is accessible to it, is actually stored in the memory, or whether it is computed each time it is requested. Similar independence of data representation is an essential feature in "structured programming", and several efforts are being made towards

incorporating such ideas into programming languages [6,7,15] . From the viewpoint of formal description it does not matter whether this independence of representation is a run time feature, as it typically is in data base systems, or whether program compilation is allowed to optimize object programs by making use of actual representations, as it should be done in programming language implementation.

5. Abstract Syntax for a Behavioral Model.

The discussion in the previous section can be made more precise by describing a system of automata as a structured object. Then we have an abstract syntax both for the language to be defined and for the objects in terms of which we wish to define its semantics.

A system of automata is a dynamic object which is modified during its operation. New automata can be added to the system during execution, and they can be in different stages of execution at different points of time. Initially, the system consists of only one automaton, which is in its initial stage of execution. This initial system is given as the semantic attribute of a program.

As terminal sets of an abstract syntax for the behavioral model we have the set of primitive automata P , the set of primitive schemes for automata S , and the set of operation identifications of automata I .

Let us now consider objects with the following structure:

s :System $\to$ SYST⟨a:Automaton*,(p:P*)⟩
a:Automaton $\to$ AUTOM⟨sub:Scheme*,e:Expression*,op:Operation*⟩
s:Scheme $\to$ SCHEME⟨f: Formal*, sub:Scheme*, e:Expression*,op:Operation*⟩
op:Operation $\to$ OPER⟨s:Scheme,(id:I)⟩
f:Formal $\to$ FORMAL⟨ ⟩
e:Expression $\to$ PRIM⟨(p:P)⟩
$\to$ AUT⟨(a:Automaton)⟩
$\to$ FORM⟨(f:Formal)⟩
$\to$ NEW⟨(s:Scheme),par:Expression*⟩
$\to$ NEWPR⟨(s:S),par:Expression*⟩
$\to$ OP⟨target:Expression,(id:I),par:Expression*⟩

Here a System consists of a finite number of Automata and primitive automata $p \in P$. Each Automaton consists of a sequence of Expressions, which are "evaluated" during initialization, of Operations, and of Schemes for the construction of further Automata. A Scheme is similar to an Automaton, except for having Formals for parameterization. An Operation is simply a Scheme together with identification $id \in I$. An Expression can be an automaton, a Formal, a scheme together with parameters, or a request for an operation of a target automaton. We shall require that every Formal appearing as a secondary component is a primary component of an enclosing Scheme. Also, the identifications of Operations within one automaton and scheme are required to be distinct. (Except for primitive automata and schemes, these requirements could easily be expressed in terms of attributes associated with the syntax.)

The behavior of a System can now be defined as follows. Initially a System consists of exactly one Automaton and of no primitive automata $p \in P$. Its behavior is defined as the sequential evaluation of Expressions e[i] of this automaton.

The evaluation of an Expression gives an automaton as a result, and is defined as follows:

> If the Expression is an AUT (or a PRIM), the result is the Automaton a (or the primitive automaton p).
>
> No Expression of the form FORM will ever be evaluated, since such an Expression cannot appear as an e[i] of an Automaton.
>
> If the Expression is a NEW (or NEWPR), a new Automaton (or new primitive automaton) is created using the scheme s and parameters par[i], and the initialization of this new automaton will be started. This new automaton is attached to the current System as another component a[i] (or p[i]), and it will itself be the value given by the evaluation of the Expression.
>
> If the Expression is an OP, the target and all par[i] are at first evaluated in sequence. Then the id-operation of the target is evoked using parameters par[i]. The value given by the OP-Expression will be the value obtained as a response from this operation.

An operation of an automaton is evoked as follows:

Each automaton is assumed to have a mechanism delaying operations so that no operation is started while the automaton is being initialized, or when another operation is being performed. For primitive automata, operations will not be specified further.

Evoking an Operation on a non-primitive Automaton means creation of a new Automaton using the Scheme of that Operation and the parameters par[i] given in OP. The value given by the Operation is the value given by the evaluation of the last Expression of this new Automaton.

In order to define how new automata are created, we need some notations for constructing structural objects of their components. In general, given a production

$$c{:}Class \rightarrow PROD\langle c1{:}Class1, \bullet\ ..,cn{:}Classn\rangle$$

and some objects $x1 \in Class1, \ldots, xn \in Classn$, the expression

$$c = PROD\langle c1{:}x1, \ldots, cn{:}xn\rangle$$

will denote an object $c \in Class$ with components $x1, \ldots, xn$ according to the production PROD. (Although not indicated above, all components xi need not be primary.) As each object can be a primary component of at most one object, it is required that for each ci with a defining occurrence in PROD, the corresponding xi is not a primary component of any other object.

In addition to constructing objects of components, we need a notation for "copying" objects and "replacing" some of their secondary components by others. More precisely, given objects x, a, b, another object y is obtained of x by replacing applied occurrences of a in x by b, denoted as $y = x^{b}_{a}$, iff there is a one-one mapping $f{:}p(x) \rightarrow p(y)$ with the following properties: (i) $f(x) = y$; (ii) for each $z \in p(x)$, objects z and f(z) belong to the same nonterminal class; (iii) for each $z \in p(x)$, if z consists of components $z1, \ldots, zn$, then f(z) consists of components $u1, \ldots, un$ such that

- if zi=a is a secondary component, then ui = b ;
- else, if zi is a secondary component and zi / p(x) , then ui = zi ;
- else, ui = f(zi) ;
- the production for f(z) is the same as that for z , or, in case z has a as a secondary component, a production uniquely determined by the object classes of ui .

In other words, primary component relations and secondary component relations "within" x are copied into corresponding relations "within" y ; secondary components outside p(x) will be taken directly as corresponding components in the "copy". Notice that the last condition above prevents the use of the notation xba if ambiguities would arise from productions with similar right-hand sides. When applied occurrences of more than one object are replaced, ordered sets of the same size can be used instead of a and b .

With these notation, the creation of a new automaton can be defined as follows:

When a new automaton is created, it is attached as another component a[i] or p[i] to the current System, and its initialization is started. For a non-primitive automaton initialization means sequential evaluation of its e-Expressions.

If $|f| \neq |par|$, the creation of a new automaton is not defined.

For primitive schemes the creation of a primitive automaton will not be further specified.

Given a non-primitive scheme s Scheme with the structure

$$s = \text{SCHEME}\langle f{:}\text{Formal}^*, \text{sub}{:}\text{Scheme}^*, e{:}\text{Expression}^*, \text{op}{:}\text{Operation}^* \rangle ,$$

together with parameters par[i] , such that $|f| = |par|$, the new Automaton will be

$$a = \text{AUTOM}\langle \text{sub}{:}\text{sub}_f^{par} , e{:}e_f^{par} , \text{op}{:}\text{op}_f^{par} \rangle .$$

In other words, a is obtained of the sub-, e-, and op-components of s by replacing each (applied) occurrence of f[i] in them by the corresponding par[i] .

Having defined a behavioral model, we can use it for the semantic definition of our example language of Section 2. This can be done by associating suitable attributes with the abstract syntax, a:: given in Figure 5. The primitive automata in the model are those corresponding to integers, denoted as p_n , and those which can be created by the primitive scheme new . The former are assumed to have operations with identifications plus , times , and neg , and the latter are assumed to have operations value , and assign .

Most of the nonterminals have only one semantic attribute $\mathcal{S}$ (for semantics). In addition, the nonterminal Block has an attribute $\mathcal{R}$ (for request), and St has an attribute $\mathcal{B}$. Notice that blocks are interpreted as schemes with one operation, with identification execute .

As an example, Figure 6 gives the value of the semantic attribute $\mathcal{S}_x$ for the object x of Figure 1. For clarity, all terminal objects are repeated in Figure 6 for each occurrence.

p:Prog → PROG⟨b:Block⟩	$\mathcal{A}_p$:= SYST⟨a:AUTOM⟨sub:$\{\mathcal{A}_b\}$, e:$\{\mathcal{R}_b\}$⟩⟩	
b:Block → BLOCK⟨v:Var*, s:St*⟩	$\mathcal{A}_b$:= SCHEME⟨f:$\{\mathcal{A}_{v[i]} \mid i \in \iota(v)\}$, sub:$\{\vartheta_{s[i]} \mid i \in \iota(s) \wedge \vartheta_{s[i]} \in \text{Scheme}\}$, op:$\{\text{OPER}\langle s:\text{SCHEME}\langle e:\{\mathcal{A}_{s[i]} \mid i \in \iota(s)\}\rangle, (\text{id:}\underline{\text{execute}})\rangle\}$⟩	
	$\mathcal{R}_b$:= OP⟨target:NEW⟨(s:$\mathcal{A}_b$), par:$\{\text{NEWPR}\langle(\text{s:}\underline{\text{new}})\rangle\}^{\iota(v)}$⟩, (id:$\underline{\text{execute}}$)⟩	
s:St → COMP⟨b:Block⟩	$\mathcal{A}_s$:= $\mathcal{R}_b$	ϑ_s := $\mathcal{A}_b$
→ ASS⟨(v:Var), e:Expr⟩	$\mathcal{A}_s$:= OP⟨target:FORM⟨(f:$\mathcal{A}_v$)⟩, (id:$\underline{\text{assign}}$), par:$\{\mathcal{A}_e\}$⟩	ϑ_s := $\emptyset$
e:Expr → CONST⟨(n:N)⟩	$\mathcal{A}_e$:= PRIM⟨(p:p_n)⟩	
→ VAR⟨(v:Var)⟩	:= OP⟨target:FORM⟨(f:$\mathcal{A}_v$)⟩, (id:$\underline{\text{value}}$)⟩	
→ SUM⟨e1:Expr, e2:Expr⟩	:= OP⟨target:$\mathcal{A}_{e1}$, (id:$\underline{\text{plus}}$), par:$\{\mathcal{A}_{e2}\}$⟩	
→ PROD⟨e1 :Expr, e2:Expr⟩	:= OP⟨target:$\mathcal{A}_{e1}$, (id:$\underline{\text{times}}$), par:$\{\mathcal{A}_{e2}\}$⟩	
→ NEG⟨e1:Expr⟩	:= OP⟨target:$\mathcal{A}_{e1}$, (id:$\underline{\text{neg}}$)⟩	
v:Var → ATOM⟨ ⟩	$\mathcal{A}_v$:= FORMAL⟨ ⟩	

Figure 5

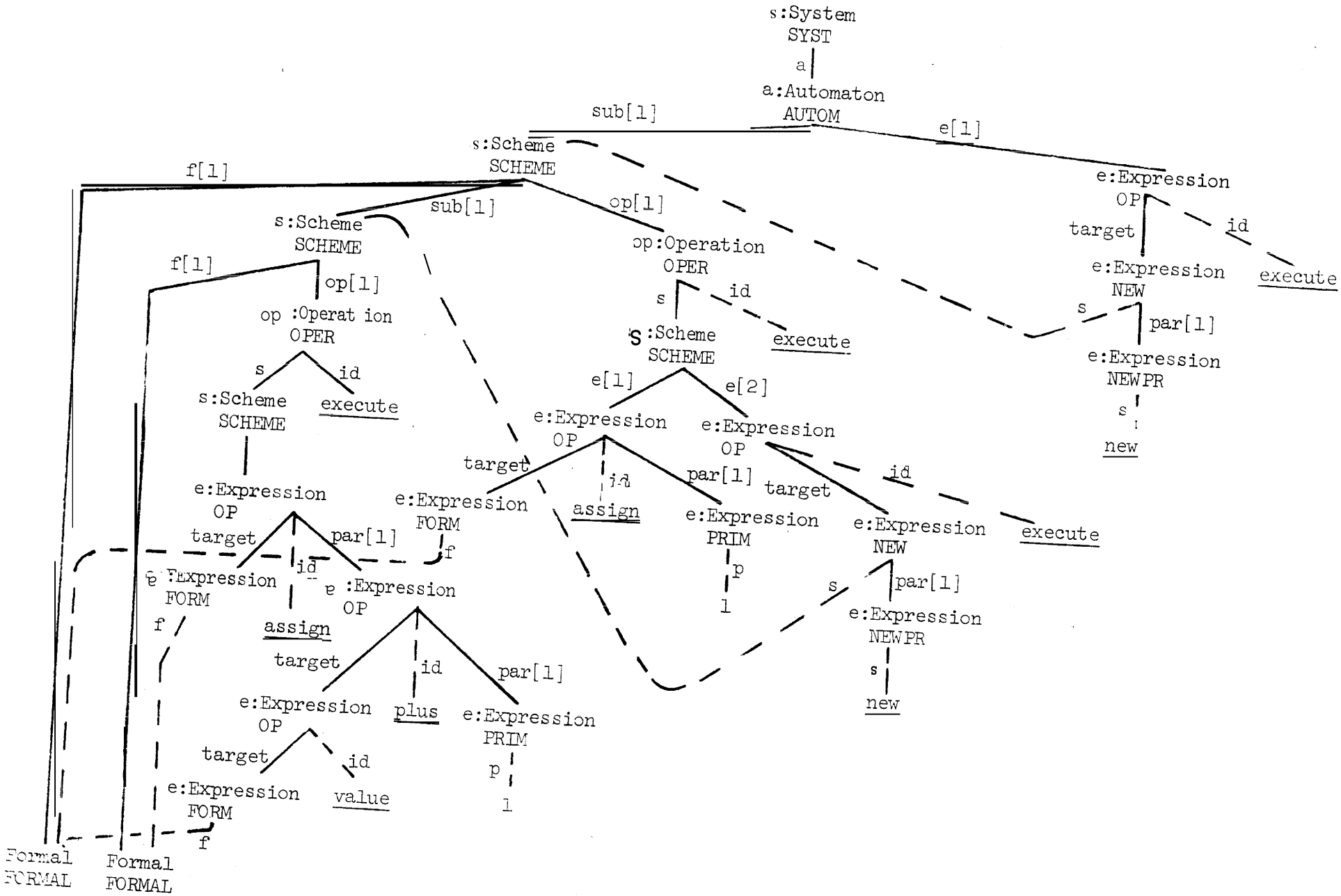
s:System
SYST
a
a:Automaton
AUTOM
sub[1]
e[1]
s:Scheme
SCHEME
f[1]
sub[1]
op[1]
e:Expression
OP
target
id
s:Scheme
SCHEME
op:Operation
OPER
e:Expression
NEW
execute
f[1]
op[1]
s
id
par[1]
op :Operat ion
OPER
S :Scheme
SCHEME
execute
e:Expression
NEWPR
s
id
s:Scheme
SCHEME
execute
e[1]
e[2]
new
e:Expression
OP
e:Expression
OP
target
id
par[1]
target
id
e:Expression
OP
e:Expression
FORM
assign
e:Expression
PRIM
e:Expression
NEW
execute
target
par[1]
f
p
s
par[1]
e :Expression
FORM
id
e :Expression
OP
1
e:Expression
NEWPR
f
assign
s
new
target
id
par[1]
e:Expression
OP
plus
e:Expression
PRIM
target
id
p
e:Expression
FORM
value
1
f
Formal
FORMAL
Formal
FORMAL

Figure 6

6. Summary and Conclusions.

Different kinds of methods and tools are being used for the definition of programming languages, depending on the intended audience and on personal preferences. Some general principles seem to have emerged, however, which can be applied independently of the particular choices.

One question which is often discussed, is the appropriate level of generality of definition methods. If the definition is intended to be used by mechanical aids for compiler constructian, then too powerful methods lead to undesirable complexity of algorithms. Experience has shown that no single tool has the right generality for defining all aspects of a programming language for such purposes, and the only solution is then to divide the definition into several stages, each using its own methods and tools tuned to an appropriate level of generality. On the other hand, when the definition is intended for human audience, it is not the generality that counts, but how this power can be "structured" to be used in an understandable way. In this case, it would also be a disadvantage, if too many different methods would be needed. The particular methods discussed in this paper are intended purely for human audience, and it is understood that they are much too powerful for mechanical manipulation, unless essential restrictions are imposed.

Language definitions usually consist of syntax and semantics. There is no sharp distinction between these two parts of definition, in principle; syntax is just a first approximation. In fact, what is included in syntax, is usually strongly influenced by the particular definition methods selected. For instance, features not describable by a context-free grammar are often left to semantics. On the other hand, w-grammars are an example of a syntax formalism which is powerful enough to include the whole definition in syntax, if one sc wishes.

For a mechanical processor, syntax determines a set of character strings, and semantics associates meanings to a subset of syntactically correct strings. For a human reader, syntax has another important function in providing a basis for abstraction. In fact, it is this

function, not the generation of syntactically correct strings, that can justify starting language description for humans by syntax. An abstract conception of programs is essential for understanding a programming language, and this seems to be the natural starting point in designing, learning, and using a language. While it is a translator's task to inspect arbitrary programs and to determine their meanings, a user proceeds in the reverse direction: he has an abstract -program in mind and wants to express it in the language.

Unfortunately, it seems impossible to provide a suitable abstraction of a complex language by its string syntax. In Algol 68 Report, for instance, this abstraction becomes quite complicated because of several aspects of external string representations, such as coercion rules, equivalence of modes, and the correspondence between defining and applied occurrences of identifiers and operators.

A natural conclusion of the above is to separate the description of the abstract structure of programs from the string syntax of the language. This means that language description should be started with an abstract syntax. In addition to providing suitable abstraction, an abstract syntax divides language properties in a way which seems more natural than the classical separation of syntax and semantics. Properties dealing with character string representations of abstract programs can be called surface -properties, while the rest will be deep properties. Again, it depends on one's abstraction, how different -properties are classified into these categories. In any case, such a classification should not be based on limitations in the tools used for the definition, but on an intuitive understanding of what are the essentials in an abstraction of programs.

As an example of a general formalism for abstract syntax, we have proposed a system of structural productions which allows both defining and applied occurrences of objects. Surface properties can then be defined by attributes associated with these productions.

Since abstract programs may still be too complex for direct semantic definition, another level of abstraction, called a behavioral model, is proposed. In some sense this corresponds to the idea of compiled programs, and the same behavioral model might well be used for the semantic

definition of several languages. This level divides the deep properties of a language into two subsets: those which are inherent in the behavioral model, and those which are associated with the mapping of abstract programs into this level. Again, there is no unique way to determine this classification. Being an abstraction of compiled programs in execution, the behavioral model should reflect what is considered conceptually most important in such processes, and it should suppress practicalities like those imposed by the bounded memory size of real computers.

We have presented a simple example of a behavioral model. It is based on envisaging program execution as creation and interaction of a collection of automata. This view is extended to primitive values, which are also considered as independent automata. Creation of new variables corresponds to creation of new automata, based on some primitive schemes for automata. Including primitive values (together with the operations allowed on them) and primitive schemes for variables in the behavioral model, means that at least some aspects of the concept of type will be inherent in the behavioral model. However, "static" aspects of type consistency can best be discussed as theorems concerning the mapping between abstract programs and the behavioral model. As another example we notice that memory allocation strategies are not reflected in any way by the behavioral model. All automata will continue to belong to the system indefinitely, once created, even if they remain "passive" and inaccessible to the rest of the system. Again, sufficiency of certain memory allocation strategies could be stated as theorems about the mapping of abstract programs into the behavioral model.

No attempt has been made in the present paper to define a complete programming language using the methods and principles advocated. Therefore, the formalisms and models proposed should only be taken as hints and fragments to be considered when defining a complex language. In particular, we believe that the general principles presented would help to structure the complexity of a programming language definition in a more manageable way.

References

[1] Brinch Hansen, Per, Operating System Principles. Prentice-Hall, 1973.

[2] _____, "Concurrent Programming Concepts," Computing Surveys 5, 223-245, December 1973.

[3] Dahl, O.-J., B. Myhrhaug, and K. Nygaard, "The Simula 67 Common Base Language," Norwegian Computing Centre, Oslo, 1968.

[4] Dennis, Jack B., "On the Design and Specification of a Common Base Language," Report MAC TR-101, Massachusetts Institute of Technology, June 1972.

[5] Ellis, David J., "Semantics of Data Structures and References," Report MAC TR-134, Massachusetts Institute of Technology, August 1974.

[6] Geschke, C. M., and J. G. Mitchell, "On the Problem of Uniform References to Data Structures," Proc. International Conference on Reliable Software, ACM, 31-42, 1975.

[7] Greif, Irene, and Carl Hewitt, "Actor Semantics of PLANNER-73," Second ACM Symposium on Principles of Programming Languages, 1975.

[8] Hoare, C. A. R., "Proof of Correctness of Data Representations," Acta Informatica 1, 271-281, 1972.

[9] _____, "Monitors: An Operating System Structuring Concept," Comm. ACM 17, 549-557, October 1974.

[10] Johnston, J. B., "The Contour Model of Block Structured Processes," SIGPLAN Notices 6, 55-82, February 1971.

[11] Knuth, D. E., "Semantics of Context-free Languages," Mathematical Systems Theory 2, 127-145, 1968. See also "Correction," vol. 5, 95-96, 1971.

[12] Kurki-Suonio, R., "An Approach to Data Structures," Unpublished paper, circulated in IFIP WG 2.2 Bulletin 3, March 1970.

[13] Landin, P. J., "The Mechanical Evaluation of Expressions," Comput. J. 6, 308-320, January 1964.

[14] _____, "A Correspondence Between ALGOL 60 and Church's Lambda-Notation," Comm. ACM 8, 89-101, 158-165, February/March 1965.

[15] Liskov, B., and Zilles, S., "Programming with Abstract Data Types," SIGPLAN Notices 9, 50-59, April 1974.

[16] Lucas, P., and K. Walk, "On the Formal Description of PL/I," Annual Review in Automatic Programming, vol. 6, part 3, Pergamon Press, 1970.

[17] McCarthy, J., "Towards a Mathematical Science of Computation," Information Processing 1962, 21-28, North-Holland, 1963.

[18] _____, "A Formal Description of a Subset of Algol," in Formal Language Description Languages for Computer Programming, (ed. T. B. Steel, Jr.), 1-12, North-Holland, 1966.

[19] Morris, James H., Jr., "Types are Not Sets," ACM Symposium on Principles of Programming Languages, 1973.

[20] Naur, P. (ed.), "Revised Report on the Algorithmic Language ALGOL 60," Regnecentralen, Copenhagen, 1962, and elsewhere.

[21] Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," Comm. ACM 15, December 1972.

[22] Strachey, C., "Towards a Formal Semantics," in Formal Language Language Description Languages for Computer Programming, (ed. T. B. Steel, Jr.), 198-220, North-Holland, 1966.

[23] _____, "Fundamental Concepts in Programming Languages," NATO International Summer School on Computer Programming, Copenhagen, 1967.

[24] van Wijngaarden, A., et al. (ed.), "Revised Report on the Algorithmic Language ALGOL 68," Supplement to ALGOL Bulletin no. 36, 1974.

[25] Walk, K., et al., "Abstract Syntax and Interpretation of PL/I," Technical Report TR 25.082, IBM Laboratory Vienna, June 1968.