## ABSTRACT

REL, Rapidly Extensible Language System, permits a variety of languages to coexist within a single computer system. Here the term "language" is understood to include a particular data base. New languages may be defined by constructing a new base language with its syntax and semantics, by extending the terminology from a given base level in order to reflect specific concepts, or by associating a given base language with a certain data base.

REL consists of an operating environment, a language processor, and the set of currently defined languages. The structural properties of these languages which determine the characterization and organization of the language processor are described. In particular, representation and manipulation of syntax and semantics are discussed, the mechanism of language extension is outlined, and the concept of a generator is introduced.

## I. INTRODUCTION

Language plays a twofold role. For an individual, or a group of individuals with some common interest, it establishes a framework within which to express the structuration of their experience and conceptualization of their environment. In a social organization it provides the conventions through which these individuals or groups exchange and relate their views. In this second role, language facilitates communication between communities with divergent interests. In its first role language supports the creative process within a given community. It becomes highly idiosyncratic and dynamic in nature as the community, or individual develops distinctive and specific concepts, and continuously reconciles them with further observations of its environment.

In such a community, the computer functions as an external memory which allows efficient and rapid presentation and organization of its stored information according to the various concepts developed. Since these concepts are expressed in a highly specific language, one must be able to converse with the computer in that very language. REL, a Rapidly Extensible Language System, is a conversational computer system designed for these purposes [1]. REL provides a community with a base language suitable to its own interests. As the community develops the conceptual structure which deals most efficiently with its environment, it constructs recursively from the base level a hierarchy of new terms or adjusts them. Since the conceptual structure is determined by observations of the environment (the "data"), so is the language. Language and data thus become closely interrelated. If chosen appropriately, the base language will remain invariant and all conceptual changes will be reflected in its extensions.

REL is designed to support a large number of diverse groups. As a consequence, it must be able to handle a large variety of languages. Efficiency considerations, as well as the necessity for easy formation and extension of a particular language suggest that a single processor be provided which deals with all the implemented languages. In order to determine the precise nature of the language processor we must develop a structural description of language. This description, in turn, will spell out the detailed organization of the language processor. It is these questions that the present paper will concern itself with.

## 2. LANGUAGES AND LANGUAGE PROCESSOR

We shall base our structural description of a language on the formalism presented earlier by F. B. Thompson [2,3]. It postulates a one-to-one correspondence between the syntactic and semantic aspects. A language refers to some domain of discourse consisting of objects and relationships among them. One can order the objects and relationships into a finite number of sets, or "semantic categories" according to their structural properties. As a practical example, the ordering may be with respect to representation within the computer memory. There exist certain "transformations" mapping categories to categories; these deal with the structural properties of the sets and apply to any of their elements. On the syntactic level, the equivalent of categories and transformations are the syntactic classes ("parts of speech") and rewrite rules of the grammar. A particular composition of rules in the grammar (a parsing tree) corresponds to a particular composition of underlying transformations. The meaning of a sentence is the effect of a given sequence of transformations on the domain of discourse.

The language processor is designed to handle these "formal languages". Even though the majority of the languages in the system can be expected to evolve from a relatively small set of base languages, the language processor must provide for languages with diverse characteristics. Our definition of formal language spans a large variety of grammars, ranging from those that are easy to describe to others that are difficult to characterize in a concise fashion. How much of this spectrum should be covered by the language processor? In other words, how complex should its architecture be? If we push its design towards accommodating the entire spectrum, the language processor will be very inefficient in dealing with formally simple languages because it would constantly have to treat aspects pertinent to only a few complex languages. If we were to tailor the processor to efficient manipulation of languages of little complexity we would limit the expressiveness which any language within the system could attain. We chose a compromise -- a solution in which the language processor deals with those structural properties that are common to the majority of what we consider interesting languages, and which are simple to formalize in terms of the demands on computer memory, and complexity of programs. The remainder of this paper specifies and discusses these properties. On the other hand, all information regarding the present state and history of the sentence analysis is made available to any language. Languages with specific characteristics are thus allowed to perform certain steps in the analysis, and change the status of the analysis, on their own.

The composite of syntactic rules and underlying transformations is a "language structure". Language is the combination of the language structure and a particular data base with objects and relationships. The language processor deals with a language only in terms of its structure and is entirely divorced from the data. The language itself, through its transformations, is responsible for carrying out all the manipulations of its data.

## 3. ANALYSIS OF A SENTENCE

The close correspondence between the syntactic and semantic aspects of a language suggests a syntax-directed analysis. The syntactic analysis of a sentence provides clues as to the semantic transformations to be applied, their combination and sequence. Thus sentence analysis proceeds in two major stages, syntactic analysis and semantic interpretation.

The syntactic analysis itself consists of three phases performed in succession: (1) parsing, (2) feature tests, and (3) syntax completion. The first examines the phrase marker accumulated so far in order to determine whether a new rule of grammar can be applied to some portion of it. The second phase is suggested by the fact that rich languages may include a large number of structural categories and, consequently, a high number of grammar rules. It is often possible to establish categories whose structural properties differ only in minor respects, and to group them into more comprehensive sets. For example, consider the singular and plural forms of a noun. Parts of speech will denote such sets of categories, e.g. "noun"; a part of speech may then be qualified by "features" (N. Chomsky [4]) according to the specific characteristics, e.g. singular or plural. Furthermore, elements in a given category must often by subcategorizied in terms of their analytical characteristics. Take as an example the precedence rules governing the sequence of arithmetical operations on numbers. Features can serve this purpose as well.

As a consequence, a rule of grammar is expressed in terms of parts of speech, and may be assigned subrules operating on features. This offers two distinct advantages. First, the number of rules, and with it the amount of searching necessary, is reduced considerably. Second, many rules need not distinguish among all the categories in a set so that the total number of grammar rules and subrules is less than the number of rules had no features been employed. In many practical cases, subrules assume an extremely simple form. If a feature can be expressed as a binary choice, then for all features being mutually independent the subrule may be formulated as "the grammar rule applies if, for each of its parts of speech, all features in a given list hold and all in a second list do not hold." In other words, the subrule may be expressed in form of two bit maps per part of speech, one identifying the features which must hold, the other specifying those which must not hold; features not referred to in either map are ignored.

If the subrule assumes a more complicated form, the language provides an explicit program, the "syntax completion" routine, to accomplish the analysis necessary. Such a program may also be needed to perform aspects of the syntactic analysis not covered by the language processor. Indeed, each rule has its syntax completion part to determine the syntactic portion of the result, possibly on the basis of its arguments.

A node in the phrase marker denotes either a "phrase" or a function symbol. A phrase consists of syntactic information (part of speech and features), and a semantic interpretation which for the time being we assume to describe a particular element in the category denoted by the syntax. In a string matched by a rule of grammar, the phrases constitute the arguments of the corresponding rule while function symbols only serve the purpose of identifying the rule. For example, in the string $\underline{N} + \underline{N}$ ($\underline{N}$ = part of speech "number") the $\underline{N}$ constitute the arguments of the rule $\underline{N} \longrightarrow \underline{N} + \underline{N}$[1], and "+" is a function symbol. Thus syntax completion and semantic transformation are functions mapping categories to a category, and individual objects to an object, respectively. The syntax completion furnishes the syntactic portion of the resulting phrase, the transformation determines its interpretation.

In many languages, the rules will be context-free, that is, of the form $a \longrightarrow b_1 \ldots b_n$. For more interesting languages, we may expect the grammar to include general rewrite rules of the form $a_1 \ldots a_m \longrightarrow b_1 \ldots b_n$. Since the combination of syntax completion and semantic transformation gives rise only to a single phrase, the language must provide an

---

[1] This notation is referred to as "generative". In the remainder of this paper the terms "left-hand side" and "right-hand side" refer to this form.

individual pair for each $a_i$.

Sentence analysis consists of repeated applications of the four steps: recognition of a rule (parsing), feature subrule, syntax completion, and semantic transformation. Since the language processor accepts general rewrite rule grammars, the primary objective in parsing is to avoid any redundant analyses, that is, repetitions of portions of the analysis. This is achieved by means of a very elegant algorithm due to Martin Kay and adapted to our purposes. Basically, parsing is single pass, right-to-left[2], bottom-to-top; all analyses of a sentence allowed by the grammar are produced.

The details of the algorithm cannot be discussed here. Let us briefly indicate that it achieves its objectives by means of dummy nodes; the connections between these nodes carry the phrases or function symbols. This is shown in figure 1. The analysis at any stage is described in terms of a directed graph. The parser explores all possible paths through the graph from a given node to its right before advancing to the next node to the left. The result of a rule is inserted as a new connection, or, in case of a general rewrite rule, as a sequence of nodes and connections. If several phrase markers develop they cannot be distinguished and are only recognized by the fact that more than one successful analysis is observed. Likewise,

---

[2] The parser organization is symmetric and could easily be changed to left-to-right direction.

parsings that did not contribute to the final analyses appear in the graph in the same way as those that did contribute.

So far we implied that syntax and semantics for a given rule are performed simultaneously; syntactic and semantic analysis proceed "in parallel". A rule may not only fail to apply on syntactic grounds but on semantic grounds as well. Its arguments may not map into any object; the path matched by the rule is meaningless. As a consequence, by simultaneousl considering the semantic aspects the size of the parsing graph may be kept considerably below what it would otherwise be, with concomitant reduction in parsing time. On the other hand, spurious parsings may not be recognized immediately but only after they participated in the analysis to considerable extent. The corresponding transformations may often be complex and time-consuming, especially when manipulating large-size data bases. Many of these parsings are finally excluded on syntactic grounds. Unnecessary execution of such transformations can be avoided by postponing the semantic analysis until the syntactic analysis has been completed; in this case syntactic and semantic analyses are performed "serially". The decision as to which alternative is the appropriate one under given circumstances is made by the language itself; it notifies the language processor of its choice.

A statement entered into the system must be converted into a form suitable for analysis. In turn, the result of the analysis must be returned to the user in intelligible form. Most languages will wish to treat some aspects of input and output in a specific way. For example, pre-editing of the statement, or substitution of lexical items may be desired prior to analysis. On output, the language may examine the parsing graph for successful analyses, treat structural ambiguities appropriately, or initiate the execution of postponed semantics. As a consequence, input and output routines are part of a language, aside from some basic services provided by the system.

Fig. 2 summarizes the basic organization of the language processor as discussed in this section.

## 4. COMPILATION AND LANGUAGE EXTENSION

When postponing the semantic analysis, the language processor must "compile" information during the syntactic analysis which enables it to perform the appropriate semantic transformations in the correct sequence. Since the syntactic analysis reflects the individual transformations, and the sequence in which they are to be combined, the compiled information contains a list of transformations in precisely the order in which their corresponding rules applied. Transformations require arguments, hence the compiled information must also contain a list of those.

A list of transformations associated with a phrase marker represents that portion of it whose semantic analysis must still be performed. In the example figure 3, this portion is marked by broken lines. Now consider the phrase marker in terms of the semantic analysis alone. By removing all broken lines and the nodes they connect to, one obtains the present status of the semantic analysis. The remainder of the analysis must, clearly, be performed on all those symbols which do not have an ancestor in the tree, excluding, of course, function symbols since they do not participate as arguments. Because transformations expect their arguments in left-to-right order, the phrases are listed in that very order. Figure 4a shows the compiled information for the given example.

After a step in the semantic analysis has been carried out, the compiled information must reflect the new status of the analysis. Thus in figure 4b transformation $T_/$ has been executed, "adding" node $\underline{N}_6$ to the phrase marker. Correspondingly, $T_/$ has been removed from the information while phrases $\underline{N}_3$ and $\underline{N}_4$ have been substituted by phrase $\underline{N}_6$. Figures 4c,d illustrate this process through the subsequent two steps. The semantic analysis is completed when no transformation is left; the list of arguments has been reduced to a single element, the result. The arguments of a particular transformation are identified by their total number, and their position in the list of arguments. These specifications remain unchanged during successive steps because of the strict right-to-left order of the analysis.

If no spurious parsings were ever found during syntactic analysis, and the possibility of structural ambiguity could be excluded, the language processor would have to maintain only a single compiled information and update it for each rule which did not fail on syntactic grounds. Since usually that is not the case, each phrase in the parsing graph carries, in place of its interpretation, the compiled information reflecting the analysis which gave rise to it and which, if executed, delivered the actual interpretation of the phrase. Whenever a rule applies, the resulting phrase receives a new compiled information derived from that of each of its arguments, and the characteristics of the present rule.

Our discussion so far did not distinguish between grammar rules constituting the base level of the language and those which are its extensions. The base level, being the invariant portion of the language, cannot be changed in the conversational mode. Its syntax completions and transformations are computer programs which determine the structural categories as well as the "primitive" operations on them. Extensions are recursively constructed from the base level in conversational mode.

There are two extreme positions for handling extensions, (1) string manipulation, or (2) compilation into some basic notation, for example the base level of the language. In the first case, a defined string of symbols is always replaced by the string of symbols which constitutes its definition. Redefinition of a term effectively propagates to redefine all terms directly or indirectly derived from it. However, if the hierarchy of compounded extensions is deep there is excessive expansion of the original string with concomitant cost in parsing time. If, on the other hand, the extension is compiled down to the base level a redefinition cannot propagate. We have chosen a compromise position that incorporates most of the advantages of both: the syntactic analysis is carried out while the semantics are postponed. Hence the semantic aspects of a new term are determined by the transformations of the terms which are directly referred to in its definition, be they base level or extension terms. At the same time, this compromise position is identical to the compilation scheme introduced above, permitting use of this scheme for language extension as well.

The language processor should treat base level and extension rules alike. Indeed, one can identify the syntax completion routine and semantic transformation for an extension rule. Both are obtained by analyzing the defining expression. For example, in the extension

$$f(x,y) : x * x + 4/y$$

the syntax for the left-hand side of the new rule is given by the syntax of the phrase dominating the expression "x * x + 4/y"; syntax completion simply reproduces this portion. The new transformation coincides with the compiled information for the expression.

In general, an extension rule will again be a function on some arguments with given structural properties. In the example above, the transformation for f(x,y) operates on any pair of objects from the categories described by x and y. Hence x and y are entirely syntactic in nature; they represent specific structural categories but have no interpretation. They are the "free variables" in the definition. Variables serve two purposes in an extension.

(1) They determine the right-hand side of the new rule. The categories they represent are described by part of speech and features. The part of speech determines the sequence of symbols in the rule; for example, if x and y are number variables, the rule reads $f(\underline{N},\underline{N})$. The features specify the subrules.

(2) In the compiled information, argument list elements which correspond to variables have no interpretation. They receive the interpretation on "definition expansion", that is, before the semantics are performed on the given arguments. Therefore, they must contain some key which relates them to the corresponding argument. This key is provided by the variables which may thus be considered as "place markers". Suppose the position of the argument in the rule serves as key. Then $\underline{N}_1$, $\underline{N}_2$, and $\underline{N}_4$ in figure 4a have no interpretation and are labeled by 1, 1, and 2, respectively.

Certain decisions with regard to language extension, such as whether to reject certain definitions on the basis of their analysis, or how to deal with structural or other ambiguities, must be left to a language itself. Consequently, each language includes a base level rule which determines the result of an extension. The language processor merely performs the compilation, ensures that the new rule is stored in standard form, and controls subsequent definition expansion. A language may also employ the extension mechanism if it wishes to avoid the use of a lexicon, and instead enter the referent words identifying objects in its universe of discourse in the form of a grammar rule. In this case each character must be considered a function symbol.

## 5. GENERATORS

We notice that general rewrite rules and definition expansion have a property in common. In each case a list of functions is given. Each function is exercised in turn, and the result of each step is utilized in a manner which depends only on the criterion governing the list. In the case of a general rewrite rule the results enter the phrase marker as a sequence of phrases, while for definition expansion they participate in subsequent steps.

This is an instance of a phenomenon known in list-processing as "generation" [5]. The general scheme of generation is shown in fig. 5. A generator can be considered a relation between two sets, an ordered set $\underline{a}$ of arguments, and a set $\underline{r}$ of results. In the course of constructing the set $\underline{r}$, the generator repeatedly selects, according to some internal criteria, an element from a set of processes $\underline{p}$, supplies it with an ordered set $\underline{i}$ of input arguments, and receives an output set $\underline{o}$ which it may simply collect, or utilize in further actions. Two cases are of special interest:

(1) For each selection j, $\underline{i}_j = \underline{a}$; each process operates on the same set of elements, namely the set of arguments for the generator. Further, each successive selection of a process is independent of previous ones; there is a list of processes which are applied in turn until the list is exhausted. Generators of this kind will be termed "operator generators".

(2) There is only a single process, that is, $\underline{p} = \{p\}$. For each selection ("pulsing"), it is supplied with a set $\underline{i}_j$ which is identical to $\underline{a}$, except that one and the same element in $\underline{a}$ is substituted on each pulsing. Again, each successive selection is independent of previous ones. Generators of this type will be denoted as "operand generators".

Under this scheme, general rewrite rules become an operator generator; each "elementary" syntax completion/transformation pair utilizes the arguments of the rule. Definition expansion exhibits some aspects of an operator generator, but the first condition ($\underline{i}_j = \underline{a}$) does not hold. However, it is the only generator of interest which does not fall into one of the two specified classes.

There is a variety of other phenomena in languages that may conveniently and efficiently be represented by generators:

Ambiguity. -- More complex languages, and certainly natural languages, permit local ambiguities within a sentence; usually these are resolved by considering a wider context within the sentence. It is those ambiguities that we wish to deal with; of course, this includes the case of an ambiguous sentence. Ambiguities arise when a grammar includes several rules with identical right-hand sides which differ in their feature subrules, syntax completions, or semantic transformations. The first two cases of ambiguity are syntactic in nature, the third one semantic. All three are described by operator generators since the same set of arguments is processed by a sequence of subrules, syntax completions, and/or transformations. Syntactic operator ambiguities can usually be resolved within limited contexts and on syntactic grounds. On the other hand, semantic operator ambiguity may render the meaning of the entire sentence ambiguous, and may be introduced deliberately in order to compare different concepts in a variety of situations. Ambiguity also arises when a transformation maps its arguments into more than one object, thus associating various meanings with a given string. In the subsequent analysis, such ambiguous interpretations of phrases will act as an operand generator. On each pulsing, the transformation of an applying rule will be provided with a new interpretation.

Numerical quantification. -- Central to many programming languages is the notion of a loop, often taking the form of a "do" or "for" statement. A given sequence of expressions is repeatedly executed, each time for a new value of one of its variables. Cumulative sum ($\Sigma$) and product ($\Pi$) are other examples of operand generators in arithmetic languages.

Linguistic quantification. -- In ordinary language we have such expressions as "all" or "some". In examining the sentence "Does some boy live in Boston?", one must consider each boy in turn until one is found which satisfies the condition, or all are checked negatively. Similarly, "what", "how many", "at least 3", etc. are handled by operand generators.

An operand generator thus refers to aggregates of objects. The individual objects are evaluated in the larger context of all or part of the sentence, and the results summarized in accordance with the particular principle characterized by the generator. Except in the case of ambiguity, such a principle must be explicitly expressed in the language. Consequently, operand generators enter the analysis through a rule of grammar. The corresponding semantic transformations differ from the ones discussed so far in that they result in aggregates rather than single elements within a semantic category. However, the previous considerations still hold if we require a transformation to produce a single interpretation for the resulting phrase. This interpretation may now be of arbitrary complexity; in the case of a generator, it may list all alternatives, or a method to construct them, and identify the particular generator.

As a consequence, the interpretation of a phrase also conveys structural properties of a language to the language processor. It may identify a single element ("data"), a generator, or compiled information, and similarly may differentiate between the base level and extensions.

At the time generators are encountered in an analysis there is often insufficient context to sum up alternatives. Hence the result of generators may again be a generator phrase. A particular generator thus propagates through the analysis until it is in a position to summarize the effect of the alternatives it introduced. As an exception, ambiguity does not summarize but excludes individual alternatives as they become meaningless in a given context. Among the operator generators, general rewrite rules add sequences of phrases to the phrase marker while syntactic ambiguity may introduce structural ambiguity. Semantic operator ambiguity, if not resolved, propagates through the subsequent analysis in form of an operand generator.

Detection of a generator indicates to the language processor that the present stage of the analysis is to apply separately to each of its alternatives. On pulsing, the generator produces a new structural description of the environment to which the analysis is reapplied. Pulsing will therefore cause the language processor to recurse at its present stage. A separate portion of the language processor directs pulsing, establishes the new environment, and controls recursion. Figure 6 shows the updated language processor organization.

A selected alternative may itself represent a generator; for example, on definition expansion or operator ambiguity a transformation may again consist of compiled information. Hence recursion may continue for several levels. Moreover, several generators may occur on a given level, for example, when several arguments of a rule are generator phrases. By accepting one generator at a time such cases are resolved into a sequence of recursions. Generators thus cause the language processor to recurse to arbitrary depth.

The use of generators raises a number of intricate issues which we cannot further pursue here. Their treatment and illustration by examples must be reserved for future publication.

## 6. LANGUAGE STRUCTURES

In order to be able to analyze a sentence in a given language, the language processor must have access to a standardized description of the language. This description consists of two major components.

1. Grammar table. -- This contains the right-hand sides of all rules of grammar. The rules are organized in the form of a "symbol tree" in which a symbol is either a part of speech or a function symbol, a node carrying a single symbol. A rule corresponds to a path starting from the top; the bit maps identifying its feature subrules are attached to the last node in the path. The tree arrangement matches the parsing strategy; as the parser advances or backtracks in the parsing graph it performs identical actions in the symbol tree.

Since parsing and feature testing constitute major functions of language processor, it is advantageous to retain the grammar table in high-speed memory during the entire analysis of a sentence. Even if the grammar table is or becomes too extensive, base level rules should continue to reside in high-speed memory while extension rules may be relegated to peripheral storage since they rarely participate in the analysis beyond the level of the input string. In such a case, however, storage organization must be such as to minimize the number of references to peripheral storage.

2. Definitions. -- The remainder of a rule, syntax completion and semantic transformation, is generally too voluminous to be part of the grammar table. Therefore, it is maintained separately on peripheral storage, linked to the corresponding node in the grammar table. Definitions are of two kinds:

(a) Base level rule. The definition, in principle, consists of the programs for syntax completion and transformation. For rules introducing generators into the analysis, it also includes the pulser. Since operator generators cannot be introduced by a rule of grammar, they are part of the language processor.

(b) Extension rule. The analysis of the defining expression determines for the rule both the syntax of its result and its transformation. Entering a definition for an extension rule is not always trivial. Since we tolerate ambiguity, accept general rewrite rules, and permit deletion or replacement of definitions, considerable bookkeeping may be necessary to ensure that any new meaning propagates to terms based on the rule in question. Because definitions must be in standard format, the bookkeeping is a function of the language processor.

It follows that the language processor, during sentence analysis, requests considerable information from peripheral storage, some of it perhaps repeatedly. This suggests page organization of memory. However, it is important that the language processor be able to deal with pages explicitly in order to arrange the various components of a language in an optimal fashion with regard to page transfers.

Among the temporary configurations guiding the language processor are the parsing graph, and the syntax and interpretations of its phrases. Since the configurations are described by lists, the language processor demands a list-formatted work area in core memory. This area is also used to describe the environment of generators, or control the recursion.

Manipulation of data is a concern of the language of which they are part. Data structures may cover a wide range, from simple formats such as single numbers to complex ones such as hierarchical file organizations or interconnected rings. If the data are transient in nature and limited in size, they may be embedded in the list work area. In most cases, however, they must be retained on peripheral storage. Again, by controlling their arrangement on memory pages a language may be able to minimize the number of page transfers.

## 7. CONCLUSIONS

Some of the notions underlying the REL language processor were first introduced in connection with the DEACON project [2]. However, this language processor exhibits vastly increased capabilities, especially the facility of accommodating a wide variety of languages, the inclusion of, and emphasis on, language extension, and the treatment of generators. A first version containing most of the described properties has been in use under the Caltech time-sharing system since spring of 1968. Since then, it has served as the basis for the development of a number of languages, and provided us with more insight into their structural descriptions. This experience led to the revised version of the language processor which constitutes the subject of this paper. The processor is supported by a multiprogramming operating system geared to the needs of REL [1].

Of necessity, this article represents a rather brief summary. In the accompanying paper on REL English, some of the topics discussed will be illustrated by one of the more prominent applications of REL.

## REFERENCES

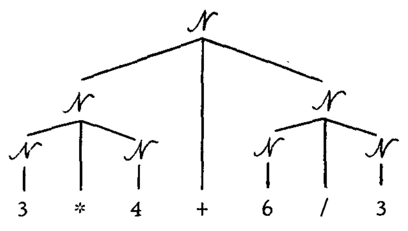
[1] Thompson, F. B., Lockemann, P. C., Dostert, B. H. and Deverill, R. S., REL: A Rapidly Extensible Language System, to appear in Proc. 24th Natl. ACM. Conf. (1969)

[2] Thompson, F. B., English for the computer, Proc. AFIPS Fall Joint Comp. Conf. 29 (1966), 349-356

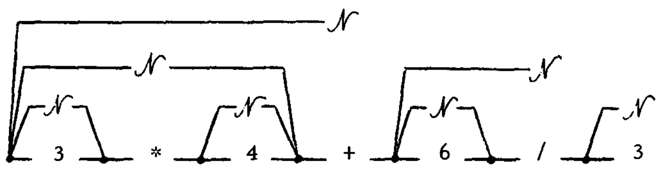
[3] Thompson, F. B., Man-machine communication, in: Seminar on Computational Linguistics, Public Health Service Publ. No. 1716, 57-68

[4] Chomsky, N., Aspects of the theory of syntax, The MIT Press, 1965, 75ff.

[5] Newell, A., et al., Information Processing Language - V Manual, The RAND Corp., Prentice-Hall Inc., 1964



(a)



(b)

Fig. 1. Phrase marker for the statement "3*4 + 6/3".

(a) Standard representation,

(b) representation as connected graph with nodes.

$\mathcal{N}$ = part of speech for "number".

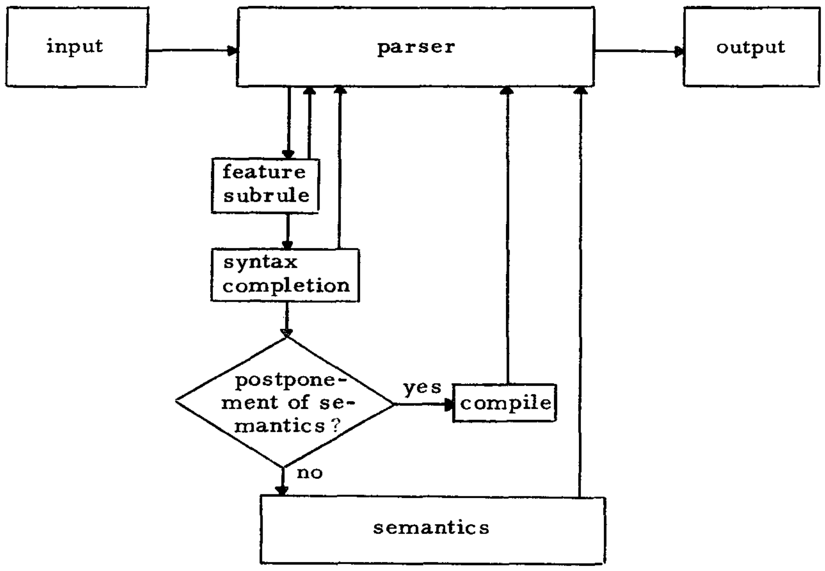


Fig. 2. Basic organization of language processor

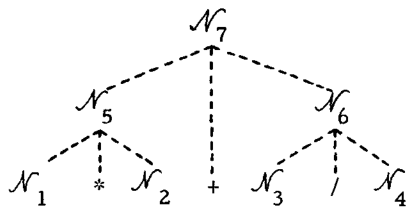


Fig. 3. Phrase marker of a simple arithmetic statement. Broken lines indicate reductions whose semantics were postponed. Subscripts are used to identify individual phrases.

| arg | trans | |
|---|---|---|
| $\mathcal{N}_1$ | $T_/$ | 3, 2 |
| $\mathcal{N}_2$ | $T_*$ | 1, 2 |
| $\mathcal{N}_3$ | $T_+$ | 1, 2 |
| $\mathcal{N}_4$ | | |

(a)

| arg | trans | |
|---|---|---|
| $\mathcal{N}_1$ | $T_*$ | 1, 2 |
| $\mathcal{N}_2$ | $T_+$ | 1, 2 |
| $\mathcal{N}_6$ | | |

(b)

| arg | trans | |
|---|---|---|
| $\mathcal{N}_5$ | $T_+$ | 1, 2 |
| $\mathcal{N}_6$ | | |

(c)

| arg | tran |
|---|---|
| $\mathcal{N}_7$ | --- |

(d)

Fig. 4. Compiled information for fig. 3. Values supplied with a transformation identify position and number of corresponding arguments.

(a) Information prior to semantic analysis,
(b-d) during successive steps of the semantic analysis.

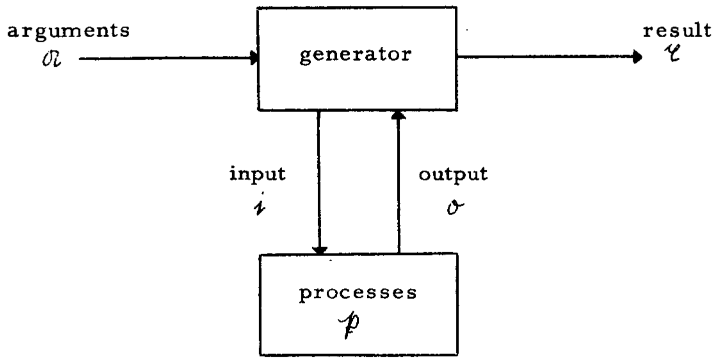


Fig. 5. Generator scheme

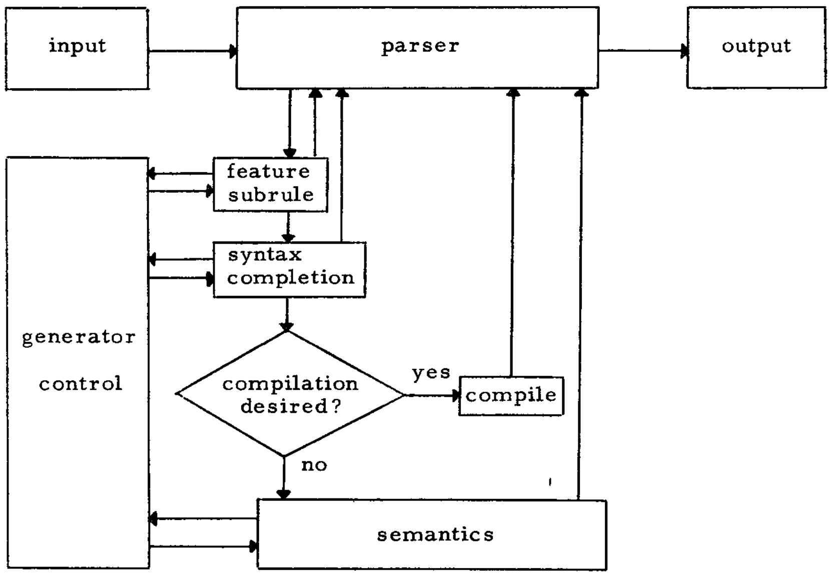


Fig. 6. Organization of language processor