UNIVERSITEIT ANTWERPEN

UNIVERSITAIRE INSTELLING ANTWERPEN

DEPARTEMENT GERMAANSE FILOLOGIE

ASPECTS OF A MODULAR THEORY OF LANGUAGE

VOL II.

Proefschrift ter verkrijging van de graad van Doctor
in de Letteren en Wijsbegeerte aan de Universitaire
Instelling Antwerpen te verdedigen door *Luc STEELS*

Promotor:
H. Brandt-Corstius

Wilrijk, 1977

§ 2. THE PROCESS THEORY

In this chapter we present a theory about language processes which is based on the modular grammar theory discussed in previous chapter.

In a first section we present a parsing system for natural language. After an introduction to the parsing problem and an intuitive overview of the model we define in full detail the representation constructs, the sort of linguistic reasoning and the control structure of the system. After that we discuss an example and shortly indicate how structures can be extracted from the result of the parsing process.

In a second section we present very briefly some ideas for a natural language producing system which consults the same linguistic information as is used by the parsing system.

§ 2. THE PROCESS THEORY

2.1. THE PARSING PROCESS

2.1.0. Introduction

In this section we present an exact model for the analysis of natural language based on the linguistic principles discussed in previous chapter. In this introductory part we define the parsing problem itself and present an overview of our system.

Normally the parsing problem for natural language is defined as the problem of how to find for a given natural language sentence the structures upon which an interpretation can take place.

However recently it has become more and more clear that this goal is not reachable simply because the input sentence itself does not contain enough information for an effective interpretation to take place . Based on the principle that the more intelligent the receiver the less explicit information you need to transmit, the information in a natural language sentence is restricted to the minimum.

So we restate the problem as follows: A parsing system extracts from the natural language sentence as much as possible information which is relevant for the interpretation process as can be done on the basis of a grammar.
The parsing problem consists then in the construction of a parsing system.

If we stick to our terminology of language phenomena and language factors, we can define the main problem in the design of a parsing system as follows. How can one observe the presence of a certain language factor. In the past two basic methods have been introduced and we want to add a third method here.

The first method is the <u>inductive</u> method (called bottom up parsing in the computational linguistics jargon). It proceeds as follows: You start from observing certain phenomena and by gradual abstraction over the phenomena you try to relate a certain phenomenon to a certain factor.

A typical notion in this context is that of a surface structure (first level of abstraction) and one deeper structure and maybe even later still a more semantic structure, etc;.

The second method is the deductive method(called topdown parsing in the computational linguistics jargon). It proceeds as follows: You start from certain grammatical expectations and you gradually translate these expectations up to a point where you are able to compare them with the language input. Notice the same ideas about small steps (but now in a reverse direction) leading from 'deep' structures to surface structures.

The third method, and the one that will be followed here, is what we will call the method of falsification . It proceeds as follows: the input elements themselves define a set of hypotheses about the factors being signalled. The system knows the relation between a factor and a phenomenon. Thus it can compute the implications of a given factor for the language situation. If these impliciations are not present, the hypothesis is falsified, else it is accepted, at least for the time being.

So, in the first methods you consider a certain phenomenon over a given input element and ask the question what pattern of my grammar applies. Suppose you have found the pattern then you ask what pattern applies next, etc.
In the falsification method a given input element tells right from the start what things it may be used for. Then you go to the grammar and ask suppose I use that input element for x, what implications does this have as regards the language phenomena over the input elements. Then you go back to the input situation and check whether it is as predicted.

In general the falsification method assumes an active grammar consultant that computes implicitions whereas the other methods assume an active representation that changes from surface to deep in small steps.

From this option follows the way in which the next main problem is approached: How are you going to bring the variety of knowledge sources relevant for parsing in motion.

In the recent history of parsing systems the discussion has been centered around the dichotomy between syntax vs. semantics directed parsers. Let us introduce these two modes of thinking briefly before we present our own position.

The first attempts (around 1960) to analyse natural language mainly from the point of view of automatic translation were mostly directed towards morphological processing and the construction of large dictionaries (see Vauquois,1976, for an overview).

The second school of thinking (around 1965) was strongly syntax based. The problem of analysis was split up in two subproblems (a) the discovery of preliminary structures representing the syntactic properties of the input, and (b) the discovery of the actual semantic structures.
In the syntax-directed parsers designed during this period, the preliminary structures represent the syntactic aspects of the sentence (in particular functional relations albeit that functional relations are sometimes indirectly represented in terms of constituent structure trees). To construct these preliminary structures a grammar in the usual sense is consulted as source of knowledge. The semantic structures are obtained by still quite complicated mappings starting from the preliminary structure.

A typical well known example of such a parsing system is the Woods' transition network parser (Woods, et.al.,1972). In this system recursive transition networks augmented with tree transforming actions and register manipulations are used to obtain the preliminary structures. To compute the semantic structures semantic rules are applied. These rules have two parts : a left part with 'templates consisting of a(syntactic) tree fragment plus additional semantic concidtions '(ibid. 2. 18) and a right part with 'forms or schemata' upon which the evaluation can take place.

The mapping of rules proceeds by matching a syntactic structure with the left part of a rule, and if successful the result is the right part.

Another example is Petrick's tranformational recognition procedure which uses a reverse transformational grammar to obtain the preliminary structures and a mapping based on patterns to compute the semantic structures stated in some predicate logic language (Petrick, 1973).

It may be of interest to point out the parallellism with the so called standard theory of transformational grammars as presented in Aspects (Chomsky,1965). The preliminary structures correspond to the deep structures in this theory and the semantic structures which in a Katz-Fodor conception often associated with this standard theory, consists of feature sequences, are obtained by some system of projection rules (Katz,1973).

The third school of thought (around 1970) which is said to perform <u>semantics-directed</u> parsing does not use the intermediary step of having preliminary structures in which functional relations or category information plays a rôle. Here one starts immmediately on the level of constructing structures which are to be used in the interpretation. A typical well known example here is Wilks' analyser(Wilks,1975)or Riesbeck's parser (Riesbeck,1976). Wilks uses templates and other forms of semantic knowledge to discover the semantic structures directly on the basis of the input. The parallel to the generative semantics viewpoint should be obvious here.

In the light of our own parser it seems that the syntax/semantics directed dichotomy can be resolved into an option for <u>all available knowledge directed parsing</u> . It is only because an hierarchical dimension was introduced in the parsing system that the question arises. We will see that this hierarchical thinking need not be the only way. In particular we will show the various knowledge sources can act in parallel and can be brought together by a supervising control structure.

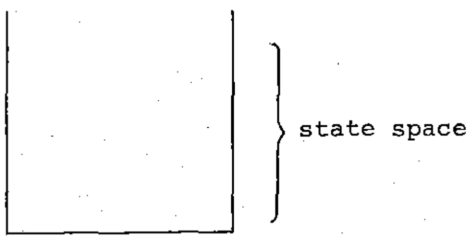
We stress that these two developments, i.e. the falsification method and the parallel application of knowledge is an immediate result of the linguistic theory presented in previous chapter, more in particular of the modular property of this theory and of the fact that the grammatical rules define a relation between a factor and a language phenomenon.

The intuitive model: the particle theory

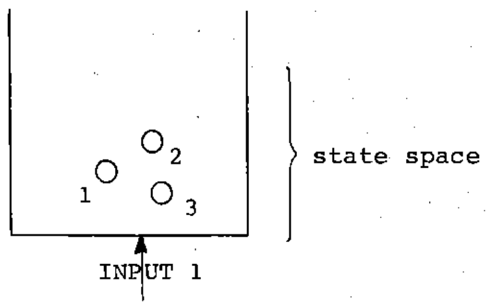
Let us now create a picture of the language process as we see it happening. (Theoretically of course. No claim is made about the psychological reality of the whole thing, although we hope psychologists may find inspiration in the model.) The description here will seem to be rather intuitive. But our aim at the moment is to evoke understanding of the general spirit and underlying ideas. The exact account up to the level of computer programs simulating the language process, as we will depict it here, will follow later.

Language can best be seen as a form of energy exchange between two information processing systems. What interests us is how the exchange takes place. Obviously there is a system which emits the energy and a system which accepts the energy. First we discuss the accepting process, normally called language understanding.

Language understanding is the evocation of a series of actions caused by the incoming energy of a language sentence. Imagine a sort of work space, which we will call the state space:

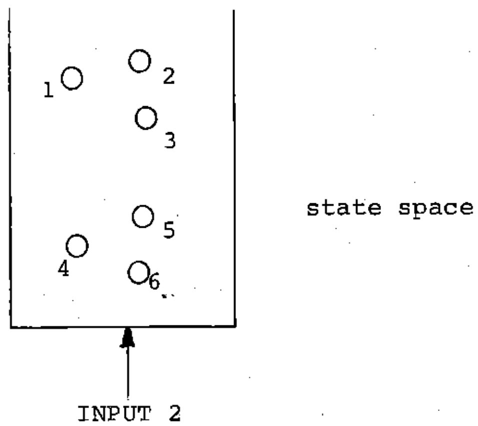


Each time an element of a language sentence comes in, it provides the energy to create one or more <u>particles</u>:



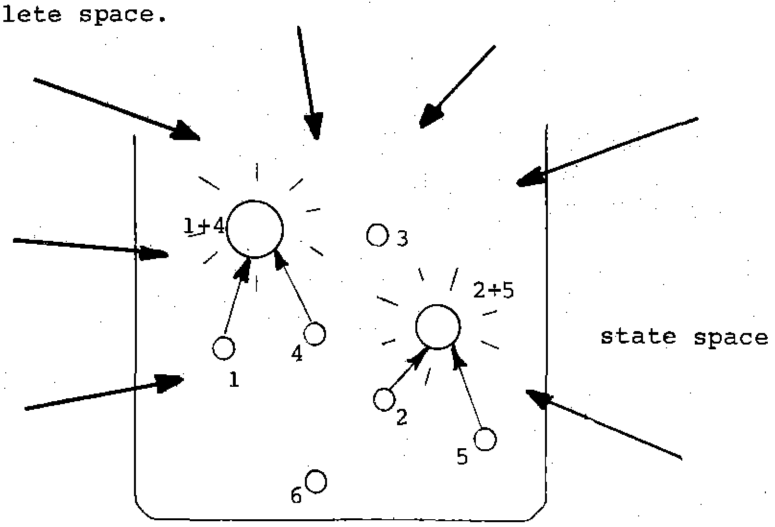
time: t1

The particles are numbered for ease of reference. The time dimension is very important. Indeed, at the next moment of time, a new pulse of energy comes in (but the old particles remain in the state space of course):



time: t2

Now comes the second sort of action : the combination of two particles to form a new one. This combination is caused by the activation of a number of forces which are resident in the state space. The word force is important here. Think about physical forces as magnetism or gravity. Although certain conditions should be met with by the particles for a force to become active, the force should be seen as a global phenomenon, present in the complete space.

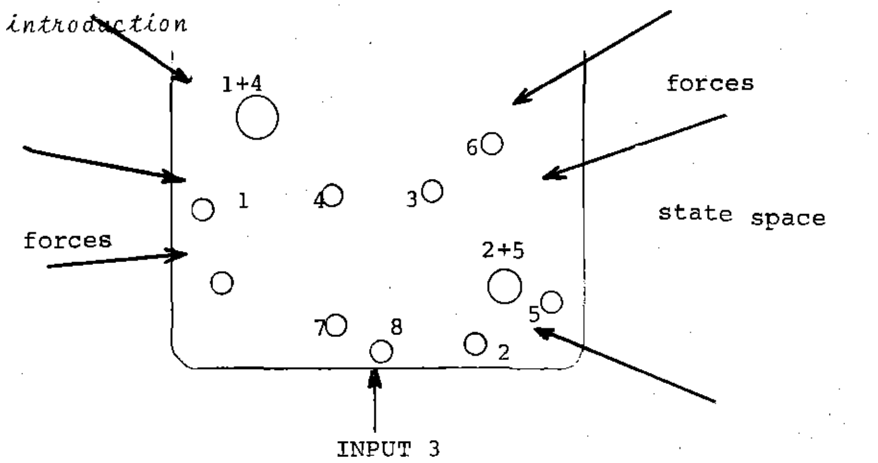


time t2"

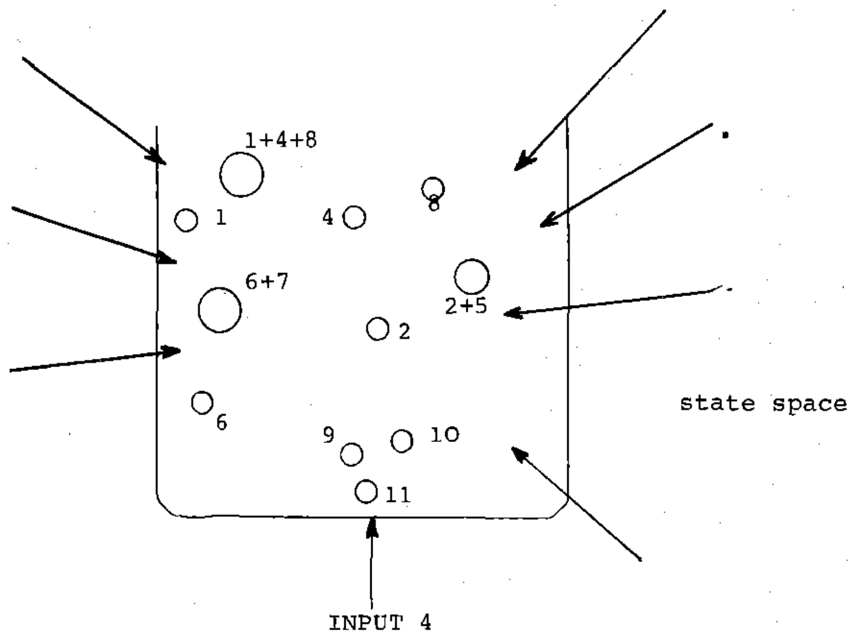
There are some general conditions for the combination of two particles, such as (i) particles created due to the same input pulse are never combin ed (ii) a particle that was combined earlier to a certain particle can later not be combined again to this particle, (iii) it is allowed however to combine the same particle with more than one other particle.

Another interesting thing is of course the investigation of the forces themselves. We will see that there are two types of forces: (i) Forces which incorporate aspects of the system of conventions that the language users agreed upon (in such a case an alternative word for force is knowledge source) and (ii) forces which incorporate results of previous actions by the system, e.g. the status of the state space as a whole is (paradoxically !) a force in the state space.

Note that the newly formed particles may still combine later with other particles which float arond in the state space. As a whole you get a regular pulse of incoming energy creating particles, and of subsequent combination processes.

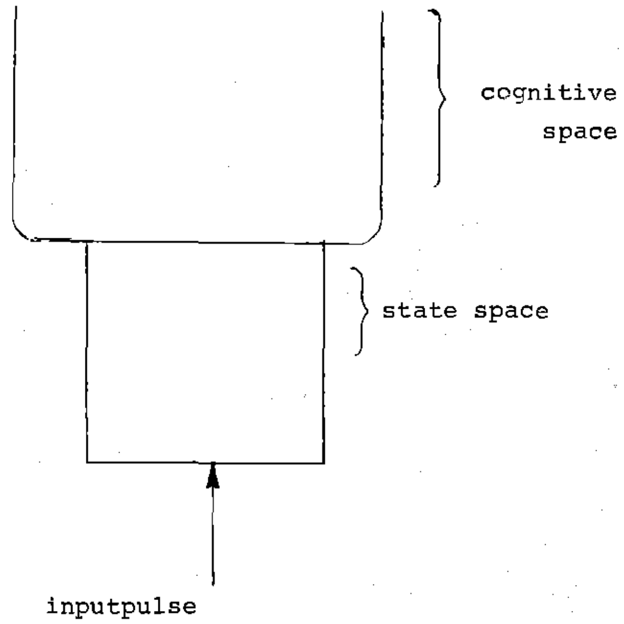


time: t3

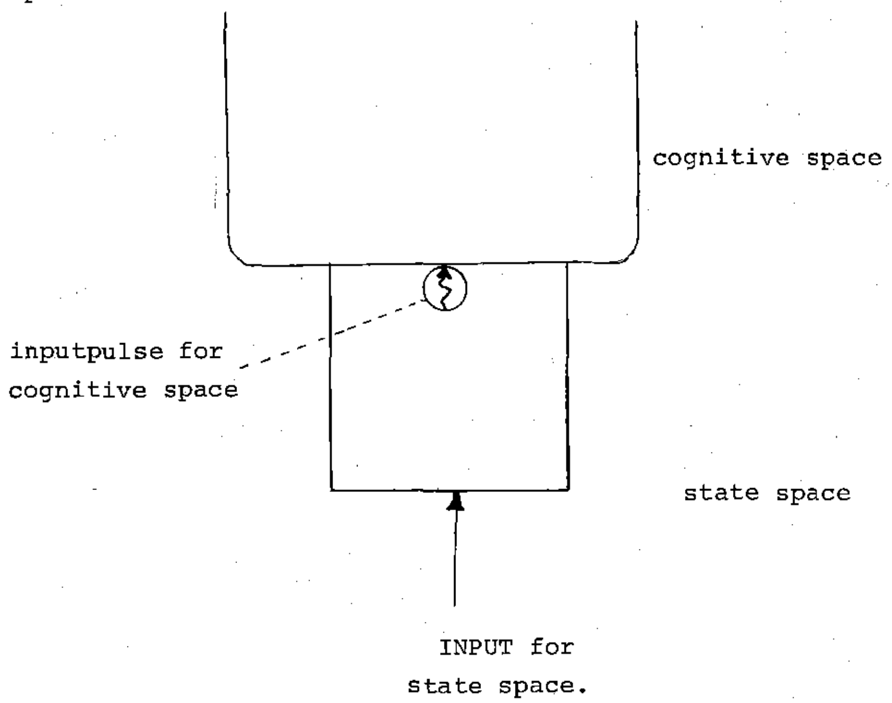


time: t4

Now comes the second part of the story. Imagine a second work space which we will call the <u>cognitive</u> space on top of the state space.



The particles travelling through the state space are now to be seen as input energy for action in the cognitive space:



Actions in the cognitive space can take the form of changing the memory structures, causing sequences of commands for physical action, causing the evocation of thought processes,etc. The particles enter a new sphere so to say, they become forces themselves.

The first type of actions (creation of particles and their combination) are called <u>analysis actions</u>. The second type (where particles themselves become forces) the <u>interpretation actions</u>. It is fruitless to assume that the two types of actions occur after each other in time, rather we say that the two phases occur in parallel, even more, although the second operates on output of the first, it turns out that the interpretation is (paradoxically)one of the forces in the analysis phase itself.

When reading this short description of the language process, the analogies with chains of chemical reactions or with interactions of physical forces will readily come into the reader's mind. We do not discourage these analogies.
too mechanistic conception of the language processing systems and the language process itself. Instead one should see it as a "living" phenomenon, in the biological sense. Typical are the goal directedness, the interaction with the environment (made up by other information processing systems), the constant evolution known as linguistic change, the maintenance of a steady state, the high interaction of the subsystems, the interconnectivity of everything, etc.. See for a general discussion of this Steels(1976,b)

A great number of questions are raised by the above description of the language process. The questions that will concern us most are:

1. what is the nature of the particles?
2. what forces are operating ?
3. what are the mechanics of each force?

These questions will be our main concern in the next paragraphs.

First we will discuss the interior details of the particles themselves (2.1.1.).then we will formalize the sort of reasoning that is embodied in the forces and how the results of reasoning interact. (2.1.2.).
The next topic is the construction of new particles: the merging process (2.1.3.). Then we discuss the general control structure of the system (2.1.4.) and give a detailed example of a complete process for one sentence (2.1.5.). We close this section by showing how structures can be extracted from the particles (2.1.6.).

Numerous examples of parsing processes will be given in next chapter when we present the experimental results.

2.1.1. Particles

We said already that a particle is a linguistic object that contains sequences of primitive information items in a structured way. The following principles will be used for the design of these information sequences:

(i) Only the information necessary to run the process is included. This implies that information which is available at other places (e.g. the dictionary) is considered to be superfluous in the particle.

(ii) We try to preserve ambiguity as much as possible, that means until it can be resolved. In practice this leads to the following options:

-a- An initial particle should be made for every possible function and for every predicate/viewpoint, i.e. for every sequence in the lexicon .

-b- Ambiguity as regards syntactic features and semantic features is preserved due to our feature complex calculus.

-c- Ambiguity as regards states in transition networks (both syntactic and semantic)is preserved.

-d- Only if due to a certain merging (on the basis of an object relation) more than one case comes out, it proves to be necessary to construct more than one resulting particle. In all other cases the combination of two particles yields only one new particle. This is a very strong result.

-e- Lexical ambiguiy which has no influence on the parsing process is preserved, even up to the level of semantic structuring . In other words some sorts of ambiguity cannot be resolved on the basis of the grammar alone.

(iii) It should be possible to compute the functional, case and semantic structures, as defined earlier, immediately on the basis of the particles. In other words no other sort of processing is allowed as interface for the semantic component.

We now define the particles in full detail. A particle contains mainly 'configurations' linked with each other. So we first define the notion of configuration.

Definition

A configuration is an n+2 tuple:

$$\langle a_1, \ldots , a_{n+2} \rangle \qquad n \geqslant 0$$

such that

a_1 is a word
a_2 is an information sequence
$a_{i+2}, \ldots , a_{n+2}$ for $i \geqslant 0$, $n \geqslant i$ other configurations

Definition

An information sequence i for adjuncts and functionwords is a 6-tuple:

$$i = \langle i1,i2,i3,i4,i5,i6 \rangle$$

such that

i1 is the hypothesis of the word under consideration; we number hypotheses according to the moment of input: INP1,INP2,...
i2 is the function name of the word for that hypothesis
i3 the state in syntactic network
according to our principle of the preservation of ambiguity we allow there to be a set of states;
i4 the state in the semantic network, also here we will allow there to be a set of states;
i5 the internal syntactic feature complex (the extension)
i6 the qual/mod/undet characteristic

An information sequence i for objects consists of a 7-tuple

$$i = \langle i1,i2,i3,i4,i5,i6,i7 \rangle$$

such that

i1,i2,i3,i4,i5 are as for adjuncts
i6 is the extension of the semantic features associated with the viewpoint of the word for the predicate in the lexicon sequence that immmediately caused this information sequence
i7 the case.

An information sequence is initially constructed on the basis of the grammar but may be changed during the parsing process. According to our first principle, we need a special reason to incorporate an item. Let us therefore now give arguments for incorporating the above information pieces and no other ones in an information sequence.

(i) The hypothesis is necessary because one word may have different hypotheses.
(ii) The function is there because we want it to be possible to extract a functional structure directly from a configuration.
(iii) The state of the function in its syntactic network is incorporated because it can be changed during parsing.
(iv) The state in the case network is only relevant if there are objects, but if so, it is obviously necessary because the state in the case network changes for every object that comes in.

For adjuncts
(v) The qual/mod/undet characteristic relevant for the semantic feature matching e.g. is incorporated because it is worked out (sometimes) by the parsing process which characteristic holds.
(vi) The internal feature complex is incoporated because it may be changed by a syntactic feature match or by features being added to it due to the send-through rule. Consistency must be kept, i.e. if a match was successful for a particular subset, then later on the same subset must be used.

For objects:
(v) For the same reason the syntactic feature complex of objects is incorporated.
(vi) And for the same reason the semantic feature complex is necessary. If an object fills a slot in one frame on the basis of a particular subset, then if a test is made whether it fits in another frame this can only be based on the same feature set.
(vii) The case itself is a necessary element for objects (except for the subject of the sentence) because it is computed during parsing time and the same initial hypothesis may later lead to different cases.

Besides a configuration a particle contains the following:

(i) The range of the configuration, i.e. from which word to which word the configuration goes,

(ii) whether the particle is open for further combination processes or not (if not we add the label LOCKED to a configuration),

(iii) the state in the syntactic network of the topword in the configuration when the reduction relation is proceeding from left to right.

In the discussion and examples (i) and (ii) will often be left out.

<u>Example</u>

```
1. ((N1) LETTER (INP4 NOM.OBJ  NIL NIL ((SING OBJ)(SING SUBJ 3PS))
   state  word   hypo  function state state
                 thesis          in    in      syntactic features
     ((THING)) NIL) )           synt.  sem.
    semantic    case             net   net
    features
```

(configuration for object with state in synt netw added on top)

```
2.  (WRITES (INP2    VERB   NIL    (W/1 FIN)   ((PRESENT)) QUAL ) )
     word    hypo  function state   state in     synt.    qual/mod/undet
             thesis          in                            characteristic
                            synt.     sem.      features
                            network   netw.
```

(configuration for adjunct)

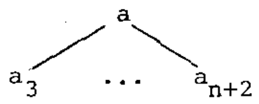
```
3. ((N5) GIRLS (INP5 NOM.OBJ NIL NIL ((BY PREP DEF TWO PLURAL))
                                            ((PERSON)) NIL)
              (BEAUTIFUL (INP4 ATT.ADJ NIL NIL NIL UNDET))
              (TWO (INP3 NUM1 NIL NIL NIL NIL))
              (THE (INP2 DETERM NIL NIL NIL NIL ))     )
```

(configuration with three depending configurations)

For the following discussion we will use schematic representations of configurations in the form of tree structures:

<u>Convention</u>

If $c = \langle a_1, a_2, a_3, \ldots, a_{n+2} \rangle$ is a configuration with $a_3, \ldots, a_{n+2}$ other configurations then we draw a tree:



We can now define the particles themselves:

<u>Definition</u>

A <u>particle</u> is a quadruple $\langle a1,a2,a3,a4 \rangle$ with

- a1 the range (i.e. from where to where in the input sequence the particle contains words)
- a2 LOCKED or NIL (keywords indicating whether the particle is no longer or still subject to combination processes
- a3 a state in a network or a set of states associated with the word in the topconfiguration of a4
- a4 a configuration.

<u>Convention</u>

As was mentioned already the range and the LOCKED/NIL will normally be omitted in the discussion.

2.1.2. The parsing predicates and their combination

Now comes the second step in the exposition: an investigation of what sort of reasoning can be used to decide whether two particles should merge or not. It is obvious that the more precise this decision process, the more efficient the parser.

It turns out that there are two main sorts of reasoning about the information in the particles, the first one is based on linguistic knowledge about the systematic aspects of the source language. The second one is concerned with the general principles of parsing that seem to govern the whole process.

Because there are many different knowledge sources available to support linguistic reasoning about language, we decided that the main problem, i.e. whether two particles should merge or not, can best be split up in a number of subproblems: should the particles merge on the basis of knowledge source x (say word order), should the particles merge on the basis of knowledge source y (say concord), etc. Once this step is taken one needs a formal model to combine the outcomes of the different consultations. We will therefore develop first of all a formal model for the combination of the results of linguistic reasoning performed by means of the parsing predicates which will be discussed in the following sections.

2.1.2.1. The combination of the parsing predicates

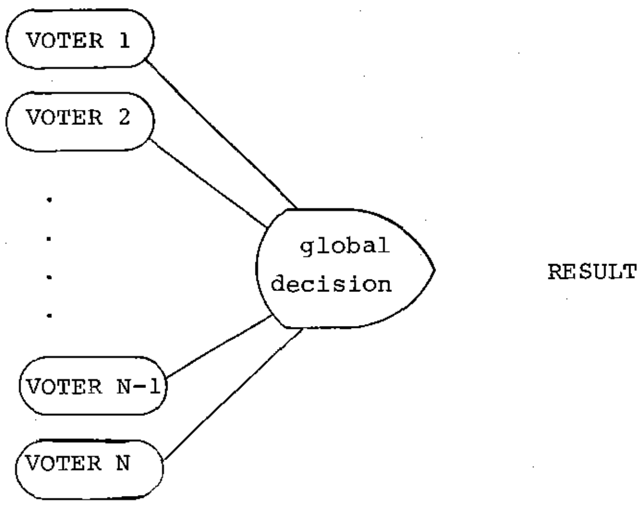
As theoretical model for the interaction of the knowledge sources we adopt a model from automata theory that was never before presented as a model for language parsing but rather as a model for doing computational geometry or solving the problem of perceiving objects and pictures ! We are thinking about perceptrons (see Minsky and Papert,1969).

(1) A set of predicates which are computable independent of each other and which all deal with a particular aspect of reality, and

(2) a decision function that brings the results of the various predicates together and thus computes the value of the predicate as a whole.

You may imagine a perceptron to be a sort of voting system where each subpredicate is a voter. The decision function is then used to compare the results of all voters and to make the final decision. Formally, it is not excluded that the decision of one voter is considered more important than that of another one, we say that the first voter has more weight than the other.

Another aspect is the treshhold which is a way to incorporate the idea that a minimum of voters must agree before the whole decision becomes positive:



Minsky and Papert define perceptrons using the notion of a treshhold and weight as follows:

Definition

"Let $\Phi = \phi_1, \phi_2, \ldots, \phi_n$ be a family of predicates.
We will say that ψ is linear with respect to Φ if there exists a number θ (the treshhold) and a set of numbers $\alpha_{\phi 1}, \alpha_{\phi 2}, \ldots, \alpha_{\phi n}$ (the weights) such that

$$\psi(X) = 1 \quad \text{iff} \ \alpha_{\phi 1}\,\phi 1(X) + \ldots + \alpha_{\phi n}\,\phi n(X) \geqslant \theta$$ " (ibid,10)

(Notice that the code for true is 1 and false 0).

Definition

"A perceptron is a device capable of computing all predicates which are linear in some given set Φ of partial predicates "(ibid,11)

Now we apply this concept to the parsing process.

The main predicate for which we want a decision true or false is this : Is it necessary to merge two particles ?
To decide on this we distinguish a number of subpredicates which we will call the PARSING PREDICATES where each subpredicate embodies a particular force. Take e.g. the predicate which applies the syntactic features match rule. This predicate checks then for a word in each particle whether there is concord between the two. If so, the subpredicate is true, else it is false. Similarly for all other phenomena.

It is important to note that each subpredicate is computed independently of the other ones.

We think that this perceptron conception of the parsing process solves the following problems:

(i) Each moment the system wants to merge two particles, all available knowledge sources can be asked to vote for or against the merging. In this way we can obtain a complete interaction of all knowledge sources on the decision and this prevents superfluous combination processes right from the start. Also we can organize the application of all knowledge sources in parallel, because each of them works independently of the others. This is certainly a fascinating idea and obviously leads to very powerful parsers.

(ii) The perceptron conception solves another great problem on which parsers currently break down, namely the problem of unreliability.

First of all there is unreliability of a knowledge source. Take e.g. semantic features testing. It is well known that any rigorous system set up to obtain consistency of semantic feature processing will break down because one can always produce semantically anomalous sentences and still be understood. The same holds for other linguistic phenomena. The sentence "he speaks not good English " is perfectly well understood, as well as "he speak not good English" and (although matters obviously become worse) "not speak he good English". But on the other hand there is a boundary of understandibility. Consider "speak good he English not".

Second there is the unreliability of the input. To say that every sentence formulated in a certain language is grammatically 100 % correct is quickly refuted by observation. E.g. there are bound to be numerous mistakes in this text due to the fact that its author is not a native speaker of the language and therefore does not know the conventions as well as someone who has been practising them all his life. Notice that the language user is not only able to understand these imperfect sentences, moreover he knows why this or that sentence is imperfect.

These two factors can in our opinion only be coped with by a perceptron conception for the interaction of the various knowledge sources, where we can attach weight to each knowledge source and where the treshhold should not necessarily be equal to a 100 % satisfaction of all subpredicates. E.g. if all but the semantic features predicate yields true , the decision function may decide that enough evidence is there to insist upon merging the two particles.

Notice that when we meet a linguistic fact that is not consistent with the linguistic description in the grammar we do not necessarily consider the grammar to be falsified by the occurrence of this phenomenon !

Having discussed the combination of the parsing predicates, we can now turn to a discussion of the parsing predicates themselves. As already mentioned in the introduction to this section there are two sorts of reasoning possible. Consequently we organize two further subsections. One about the systematics of the language and one for reasoning about the process or results about the parsing process.

2.1.2.2. Parsing predicates based on systematics of the language

The question whether two particles are allowed to merge amounts to answering the question whether a certain word say w1 in configuration c1 can act as the subordinate of another word, say w2 in configuration c2. The environment ,i.e. the other items in the configuration, may be involved in this decision as we will see and also the position of each word in its own configuration is not irrelevant. This will be discussed in § 2.1.2.3. . Here we concentrate on the two words themselves and their associated information. Consequently the predicates will be formulated on the basis of two words. We address the information sequence of a word w_k as i_{wk} and the n-th item in it as $i_{n,wk}$.

The discussion here runs parallel with the discussion of the grammatical rules, in particular there is a predicate for each rule. To make the relation between the linguistic rules and the parsing predicates explicit, we place a p-indicator before each rule, e.g. if function-of-head is a rule, then p-function-of-head is the predicate derived from it.

(1) FUNCTION-OF-HEAD and TAKING-OBJECTS

Recall the structural property that given words w1 (in configuration c1) and w2 (in configuration c2), if w1 is supposed to have a particular grammatical function f as regards w2, w2 should have a particular possible function, indicated by function-of-head (f) .

From this we extract the following predicate:

Definition

p-function-of-head : W x W → {TRUE, FALSE} is defined for (∀w) ($i_{2,w1}$ ∈ F-adj ∪ F-functw) as follows:

$$\text{p-function-of-head}(w1,w2) = \begin{cases} \text{TRUE if function-of-head } (i_{2,w1}) = i_{2,w2} \\ \text{FALSE otherwise} \end{cases}$$

Recall also that for objects the information was stored vice-versa by means of the taking-objects rule telling whether a word takes objects or not. This leads to the next predicate:

Definition

p-taking-objects: W x W →{TRUE,FALSE} is defined for (∀w1) ($i_{2,w1}$ ∈ F-object) as follows

$$\text{p-taking-objects } (w1,w2) = \begin{cases} \text{TRUE if taking-objects}(i_{2,w2}) = \text{TRUE} \\ \text{FALSE otherwise} \end{cases}$$

(2) Word order

The second property is that two words should be in a relative position as regards each other for a particular grammatical relation to hold.
We use two linguistic rules for this purpose: position (if the subordinate has the function adjunct or functionword) and object-position (if the subordinate has the function object). Consequently we will have two corresponding predicates. But first we need an auxiliary predicate.

Definition

We say that a word w_i comes before another word w_j denoted as $w_i < w_j$ if in the input sequence we have
$w_1 \ldots w_i \ldots w_j \ldots w_n \quad n > 0$ and $1 \leq i \leq j \leq n$

Definition

Let p-position : $W \times W \rightarrow \{TRUE, FALSE\}$ be defined for $(\forall w1)\ (i_{2,w1} \in$ F-adjuncts $\cup$ F-functw) as follows:

$$\text{p-position}(w1,w2) = \begin{cases} \text{TRUE if } \underline{\text{position}}(i_{2,w1}) = \text{before or undet and } w1 < w2 \\ \text{FALSE otherwise} \end{cases}$$

Definition

Let p-object-position : W x W → TRUE, FALSE be defined for $(\forall w1)\ (i_{2,w1} \in$ F-object) as follows:

$$\text{p-object-position}(w1,w2) = \begin{cases} \text{TRUE if } \underline{\text{object-position}}(i_{1,w2}) = \text{before or undet and } w1 < w2 \\ \text{FALSE otherwise} \end{cases}$$

(3) Syntactic networks

Completion automata are used in the system to regulate in a nontrivial way the mutual restrictions that occur when different subordinates are related to the same head.

An important assumption behind the use of these networks (when used in a left-going mode) is that the ranges of the unit relevant for the transitions in a network are bordering on each other and as soon as a unit is encountered that does not fit, the network is assumed to enter a final state. In this way we can discover the boundaries of word groups and it must be noted that the method works excellent.

Another nice consequence of the assumption is that the state in the network should not be incorporated in the information sequence of the topword of the combination but can be stored externally in the particle itself and be declared irrelevant as soon as the boundary of the network has been found. This is the reason why we defined such a state as being located outside a configuration.

The predicate relevant for syntactic networks is then defined as follows:

Definition

p-synt-network: $W \times W \rightarrow \{TRUE, FALSE\}$ is defined ($\forall$ w2) (syntactic-network $(i_{2,w2})$ is defined) as follows:
Let $S = s_1, \ldots s_n$ be the set of states associated with the particle of w2 , then

$$\text{p-synt-netw}\ (w1,w2) = \begin{cases} \text{TRUE if } (\exists s \in S)\ (\ \gamma(i_{2,w1},s) \neq \emptyset) \\ \\ \text{FALSE otherwise} \end{cases}$$

The second aspect in relation to syntactic networks is that a set of new states is associated with the particle. This operation is however dealt with in the section where we deal with the construction of new particles.

(4) Concord

The next predicate has to do with the syntactic feature matches based on the feature complex calculus we introduced in previous chapter.

Definition

p-concord: W x W→{TRUE, FALSE} is a function defined
(∀ w1) (w1 ∈ F-object)

p-concord(w1,w2) =
- TRUE if either
  - (i) concord ($i_{2,w1}$) = false
    or
    concord ($i_{2,w1}$) = true and
    syntactic-feature-complex of w2 matches with $i_{5,w1}$
- FALSE otherwise

(5) Send-through

The other aspect having to do with syntactic feature complexes is the phenomenon that certain features are 'send-through' to the feature complex of the head. This is again a situation where the information sequence is changed and this will be discussed in the relevant subsection.

Now comes the second series of predicates related to case.

(6) Semantic features for adjuncts

The next parsing predicate investigates whether the head of a function has the appropriate semantic features to fill a slot in a frame of a subordinate.

For this purpose it is necessary (i) to compute the semantic features that are to be satisfied by means of the viewpoint of the adjunct, (ii) to compute the semantic features that are associated to the slot filler (recall the additional complexity due to the modifier/qualifier dinstinction) , (iii) to see whether both features match, in particular whether the result of (ii) matches with the result of (i). If the result of the match yields true the predicate is true, else false.

<u>Definition</u>

<u>p-sem.feat-adju</u> : W x W → {TRUE, FALSE} is defined (∀ w1) (w1 ∈ F-adjuncts) as follows:

Let ⟨w1,w2⟩ ∈ F, p1 = predicate(w1), c1 = viewpoint (w1) and p2 = predicate (w2), c2 = viewpoint (w2) then

<u>p-sem.feat-adju</u> (w1,w2) =

- TRUE if
  - (i) either F has the modifier/undet characteristic and <u>match</u>(valuerestriction(self,p2), valuerestriction(c1,p1)) = TRUE
  - or F has the modifier/undet characteristic and <u>match</u>(valuerestriction(c2,p2), valuerestriction(c1,p1)) = TRUE
- FALSE otherwise

A side-effect of the p-sem.feat-adju predicate is that the domain of the semantic features complex of the head involved is restricted to the set of subsets satisfying the value restriction to be satisfied.

(7) Semantic networks

Next we have the predicate which consults the semantic networks: on the basis of the syntactic features complex it is investigated whether there is a transition possible.

Definition

p-sem-netw : W x W $\rightarrow \{$TRUE, FALSE$\}$ is defined
($\forall$ w1) (w1 $\in$ F-objects) as follows:
Let $S = \{s_1, \ldots, s_n\}$ be the set of states in the case networks with the configuration of w2, then

$$\text{p-sem-netw}(w1,w2) = \begin{cases} \text{TRUE if } (\exists s \; S)\ (\ \gamma(i_{5,w1}, s) = \emptyset\ ) \\ \text{FALSE otherwise} \end{cases}$$

Notice the side-effects: we can compute c, because c is associated with a transition in the network, we have a new state in the case network and , because of the feature match, a subset of the syntactic feature complex will be cut out of the domain.
This information will be of use in the construction of a new particle.

(8) Semantic feature test for objects.

The final predicate deals with the test whether the semantic features of an object are compatible with the case it wants to fill in a certain case frame.

Definition

p-sem.feat-obj: W x W $\rightarrow$ {TRUE, FALSE} is defined ($\forall$wl) (wl $\in$ F-object) as follows:

Let $\langle$wl,w2$\rangle$ $\in$ f, pl = predicate (wl) , cl = viewpoint (wl), p2 = predicate (w2) , and c one of the cases of p2, then

p-sem.feat-obj (wl,w2) =
- TRUE if match (valuerestriction(c,p2), valuerestriction(cl,pl)) = true
- FALSE otherwise.

A side effect of this predicate is the restriction of the semantic features complex of the object involved.

We have now presented predicates for all rules in the modular grammar defined in previous chapter. We now turn to reasoning based on results of the process of parsing itself.

2.1.2.3. Parsing predicates based on the process

In this subsection we present a number of forces which also help in the decision whether two particles merge but which do not use linguistic information to formulate a decision but rather information accumulated during parsing time. We feel that there are more facts to be discovered about these knowledge sources . Nevertheless the general assumptions about the parsing process which determine the sort of reasoning under discussion in this subsection already now proved to have a very strong impact on the efficiency of the parser.

Let us present these assumptions in some detail.

(i) The linearity of langauge

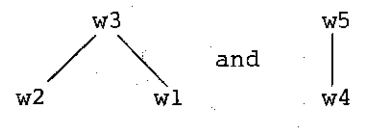
The fact that the words of a language come after each other is used by several parsing predicates (e.g. p-position). It turns out that the linear structure of language sentences can also be used to optimize the parsingprocess itself, based on the following principle:

Principle 1

A particle can only merge with another one if the range of the first particle is bordering on the range of the second particle.

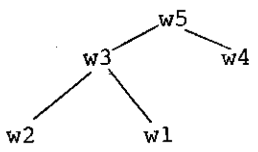
Example:

Given a sequence "wl w2 w3 w4 w5" then if there are e.g. particles on w3 and w5 containing the structures

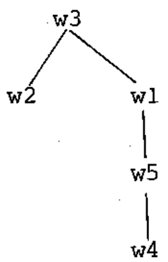


(particle 1) (particle 2)

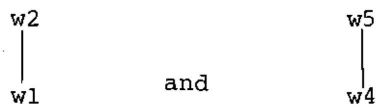
then we may consider the merging of these two which may lead to



or



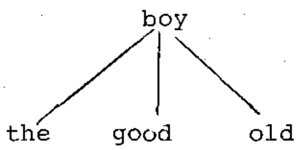
But suppose we have particles on w2 and w5 with structures



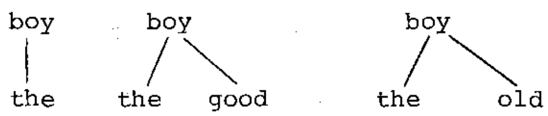
(particle 1) (particle 2)

then we will not attempt to link the two according to principle 1 because w4 is in between the ranges.

To see the value of this principle consider "the good old boy" which should result in a particle structure



But suppose we do not accept the principle, then the structures



would equally well be constructed as there is no linguistic information preventing it.

(From a formal language point of view it is interesting to note that the principle reflects the basically context-free character of natural languages !)

(2) The time dimension

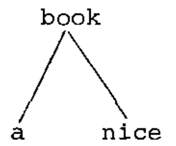
Another consequence of taking this time dimension seriously is that if a particle will be attached to one of the subconfigurations of another particle, what subconfiguration is allowed depends strongly on the time moment this subconfiguration was added to the particle. This is reflected in the following principle:

<u>Principle 2</u>

If the subconfiguration was added by a "forward merge", i.e. suppose a_j and a_i were to be merged, a_j comes before a_i, then it is not allowed to merge any new particle a_k on a_j anymore.

(Readers who think we may come in trouble with this principle should bear in mind that the parsing proceeds from left to right and therefore all possible forward merging that could be done is already done when the particle itself is subject to forward merging)

To see the point of this principle consider the phrase "he reads a nice book". Whatever comes after "book" or before "a", as soon as the structure



is created,

it is pointless to look for further combinations with "a" or "nice".
Notice that the principle does not hold for "backward merge". This can easily be understood when considering the ambiguous sentence "he saw the man in the park with a telescope".

(3) Power from structure

The final predicate to be discussed now has to do with the interrelationships of the particles:

Principle 3:

A particle with the same top as another particle but with more subconfigurations is more powerful than the other particle.

To understand this hypothesis consider the following example: "The boys sing... ". During parsing a particle will be made for "the boys", but the particle for "boys" on its own remains in the state space. Now we want to prevent that two structures are built one for "boys sing..." and one for "the boys sing..." although both of them go on the basis of linguistic information as such.

Notice that the hypothesis reflects the principle of goal-directedness which is found in most cognitive tasks: the structured objects will leave a stronger impression on our perception system than not structured ones.

Some care is needed in using the above principles. Apart from the fact that certain constructions such as coordination (which we have not yet considered) will not fall within the scope of the principles it is possible that deviations occur just as there are deviations from the linguistic predicates discussed in previous section.

Some examples of deviations: Take the expression" the author"s article". Is 'the' a determiner of 'author' or of 'article' ? According to principle 3 'the' will be considered as a determiner of 'author', and most people would agree on this. But some people would argue that at least theoretically 'the' can be considered as determiner of 'article'. Take as another example the expression 'a brighter colour than this one', where 'than' obviously relates to 'brighter' . But this is against principle 2 !

2.1.3. The construction of new particles: the merging process

Suppose that the various parsing predicates have been computed for two particles and that via the perceptron combination the final result yields positive, how is the construction of the new particle working then.

First of all we stress that this combination process is not fatal for the source particles, i.e. when a new particle is made the source particles from which it is made remain in the state space. Although the particle may be 'locked' according to principle 3 discussed earlier.

The definition of the merging process proceeds in two steps. First we define the merging of two configurations, only then we turn to the merging of two particles.
The definition of the merging of two configurations itself proceeds also in two steps. First we define the merging of two simple configurations , the so called direct merge , then we define the merging of two more complex configurations.

Definition

We say that two configurations a_i, a_j directly merge iff

$$a_i = \langle a_{1,i}, a_{2,i}, a_{2+1,i}, \ldots , a_{2+m,i} \rangle \quad m \geqslant 0$$

and

$$a_j = \langle a_{1,j}, a_{2,j}, a_{2+1,i}, \ldots , a_{2+n,j} \rangle \quad n \geqslant 0$$

then

$$\text{d-merge } (a_j, a_i) = \langle a_{1,i}, a'_{2,i}, a_{2+1,i}, \ldots, a_{2+m,i}, a_j \rangle$$

How $a'_{2,i}$ is computed from $a_{2,i}$ will be discussed shortly.

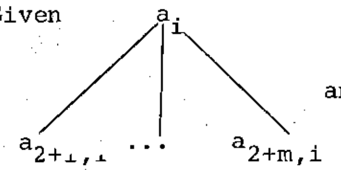
merging

Definition

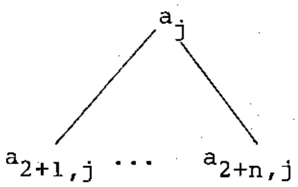
We say that two configurations a_j, a_i merge iff either d-merge(a_j, a_i) or aj merges with $a_{2+p,i}$, $1 \leqslant p \leqslant m$.
The resulting configuration is denoted as merge (a_j, a_i) .

Example

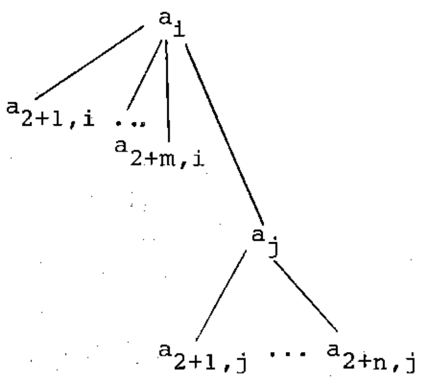
Given



and



then



$\in$ merge (a_j, a_i)

Now we can define the merging of two particles

Definition

Let $p1 = \langle p_{1,1}, p_{2,1}, p_{3,1}, p_{4,1} \rangle$ and

$p2 = \langle p_{1,2}, p_{2,2}, p_{3,2}, p_{4,2} \rangle$ be two particles

then

$p_3 \in$ merge (p2,p1) if

$p3 = \langle p_{1,1}+p_{1,2}, p'_{2,3}, p'_{3,3}, p_{4,3} \rangle$ (for $p'_{2,3}$ and $p'_{3,3}$ cf.infra)
and $p_{4,3} \in$ merge ($p_{4,2}, p_{4,1}$)

During the merging process the information in the information sequences of the respective particles are changed.

There are first of all changes in the configuration of the subordinate and second changes in the configuration of the head of the grammatical relation.

(1) Subordinate

(a) If the subordinate is an object, then side effects of the case frame application are:
(i) That we know the case;
(ii)That we know the subset of semantic features satisfying the case slot;
(iii) That we know the subset of syntactic features satisfying the case slot.
So we change the three items in the information sequence of the subordinate.
(b) If the subordinate is an adjunct we only change the qual/mod/undet characteristic.
(c) If the subordinate is a functionword no changes are necessary.

(2) Head

(a) If the head is an object,then
(i) The state of the function may have to be changed due to a transition in the networks,
(ii) Similarly the state in the case network may have to be changed on the basis of objects evoking transitions in the networks.
(iii) The subordinate may have restricted the syntactic feature complex in the syntactic feature match.
(iv) The subordinate may have restricted the semantic feature complex via the semantic features match to consult the case frames of the adjunct.
(b) If the head is not an object, then
(i) The state of the function may have to be changed due to a transition in a syntactic network,
(ii) the state in the case network may have to be changed if affected by the income of objects.

In the particle top structure we moreover change the LOCKED/NIL indicator if necessary according to principle 3 and the state in the syntactic network for the leftgoing transitions. Principle 2 is realized by hanging the indicator NIL after the information sequence of the subordinate as a sort of end marker.

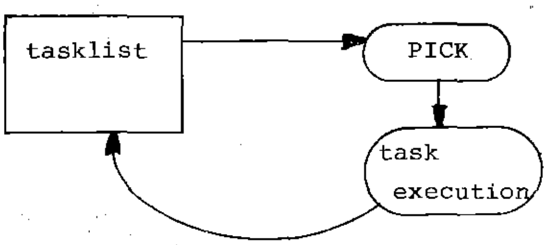
We leave a formal definition of these changes to the reader.

When a merging has taken place, the newly formed particle is investigated further to see if other combinations are possible.
To explain how this is going we present now the general control structure of the parser.

A note on the control structure

To regulate the whole process we use the concept of a tasklist and a function picking out each time the task on top of the tasklist until no tasks are left. The execution of a task may cause the creation of new tasks on the tasklist.

Schematically:



When an inputpulse comes in all particles created by this pulse are put on the tasklist. For each particle on the tasklist we try to merge with each particle associated with the word just before the range of the particle. If a merge takes place, we put the newly made particle with extended range again on the tasklist. If no merging can take place no action is undertaken. If the tasklist is empty we consume the next input inputword. If there are no inputwords left we compute the structures contained in the final particles associated with the last word of the input.

2.1.4. An example

The best way to see how a parsing process as depicted in this chapter is actually going is to consider in full detail an example. For this purpose we take one single sentence "time flies like an arrow" and although we know very well that one normally understands this sentence only as meaning "time passes by quickly" (basically because the sentence has a proverb status) we will for the sake of example assume that all possible readings should come out of the parser. These readings are by the way all produced by anyone if you explicitly ask for them.Much more examples will be given in next chapter when we discuss our experimental results.

Here are the readings:

reading (1) (the normal one) Time passes by quickly.
"Time"is an object of "flies" which is itself a predicate.
"like an arrow" is an adverbial adjunct of "flies".

reading (2) There is a particular sort of insects, called time flies and they have the shape of an arrow.
Here "time" is an adjunct of "flies", "flies" an object and "like an arrow" an adjunct of "flies".

reading (3) There is a particular sort of insects, called time flies and they love arrows.
"Time"and "flies" are as in reading (2), "like" is now the predicate and "arrow" fills a slot in the case frame of "like".

reading (4) Measure the time of a particular sort of flies, namely those which are like an arrow.
"Time" is now an imperative verb, "flies" object and "like an arrow" adjunct of "flies" as in reading (2)

reading(5) Measure the time of a particular sort of flies and do this "like and arrow".
"Time" is again imperative and "flies" object, "like an arrow" is now an adverbial adjunct of "time".

Before we can discuss the parsing process we need a small grammar which contains all the information that will be necessary for the parsing process. Let us discuss this grammar first. It is an example grammar , that means that in later experiments we do not necessarily use the same grammar.

(i) The grammar

1.1. Type object
(i) Function nom.obj (nominal object)
type : object
taking-objects: true
object-position: after

example: 'flies' as in 'to capture the flies'

(ii) function: nom.att.adj (nominal attributive adjunct)
being adjuncts formed of objects which consist of a relationword (that gets the function nom.att.adj) and an object. We will use the phenomenon of syntactic networks to make the object obligatory.
type: objective adjunct
position: after
function-of-head: nom.obj
Q/M characteristic: qual

example: 'like" as in "there are time flies like an arrow"

(iii) function: nom.adv.adj (nominal adverbial adjunct)
being adjuncts of other adjuncts which consist of a relation word (that gets the function nom.adv.adjunct) and an object. We use again the syntactic networks.
type: objective adjunct
position: after
function-of-head: verb (at least)
Q/M characteristic: mod

example: "like" in the proverb "time flies like an arrow"

(Notice that it is possible to consider only one function for nom.att.adj and nom.adv.adj but we split them up for the sake of the example.)

example

1.2. Type: adjunct

(i) function: verb
being the main verb of the sentence
type: adjunct
function-of-head: nom.obj
position: after
taking-objects: true
object-position: after
concord: true
Q/M characteristic: undet

example: "flies" in the proverb"time flies like an arrow".

1.3. Type: functionword

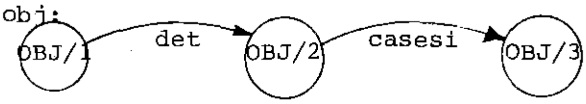
(i) function determiner (det)
type: functionword
function-of-head and position are specified via the syntactic networks associated with nom.obj
concord: true
send-through: true

example: "an" in "an arrow".

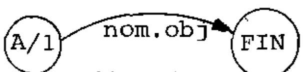
(ii) function: casesign (casesi)
type: functionword
function-of-head and position are specified via the syntactic networks associated with nom.obj.
send-through: true.
(this function is only added to make the example moreinteresting)

2 . The syntactic networks

There is one left-going network and one right-going network :
for nom.obj:



where OBJ/1 is the initial state.
and



for nom.adv.adj and nom.att.adj. FIN is the final state.

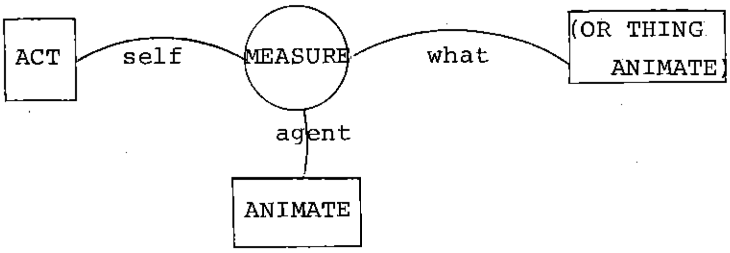
example

(3) The case frames

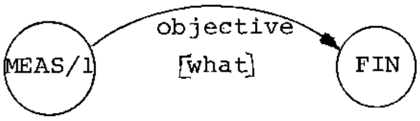
The surface case frames are only given if necessary.

-i- MEASURE

abstract case frame:

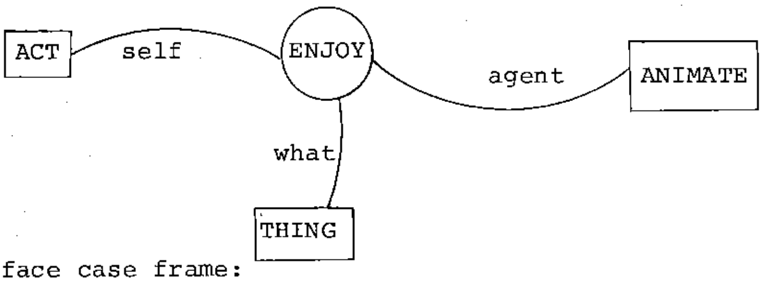


surface case frame for function adjunct and viewpoint agent



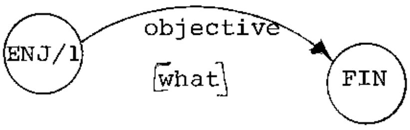
-ii- ENJOY

abstract case frame:



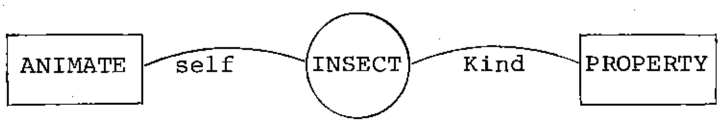
surface case frame:

for function adjunct and viewpoint agent:

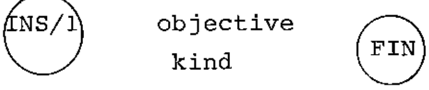


-iii- INSECT

abstract case frame:



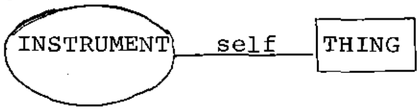
surface case frame:



example

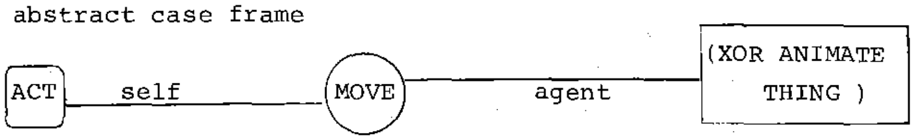
-iv- INSTRUMENT

abstract case frame:



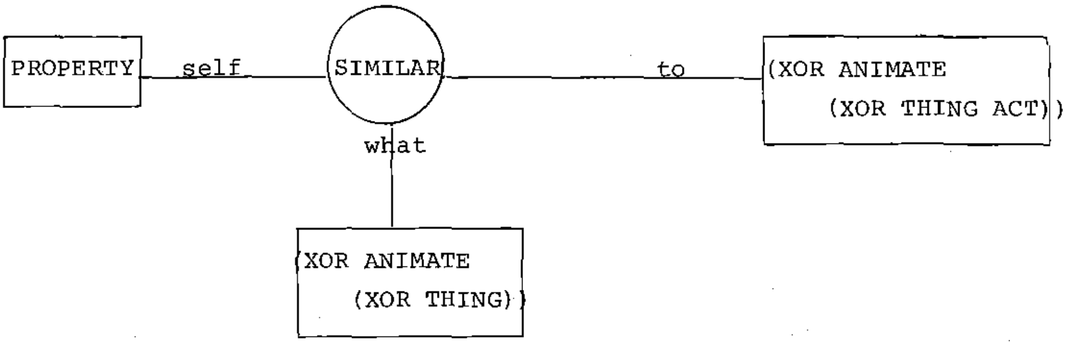
-v- MOVE

abstract case frame



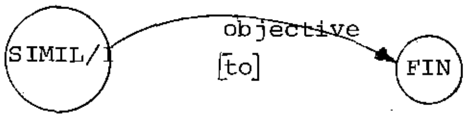
-vi- SIMILAR

abstract case frame



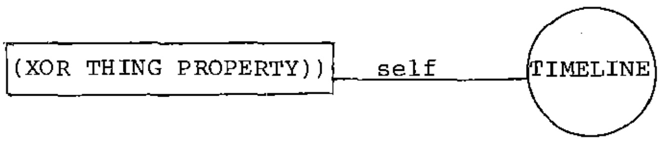
surface case frame

for function adjunct and viewpoint what=



-vii- TIMELINE

abstract case frame:



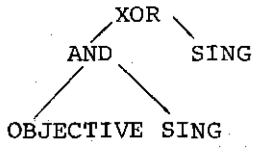
4. The lexicon

(i) AN function: det
syntactic features: SING
send-through feature: UNDEF

example

(ii) ARROW

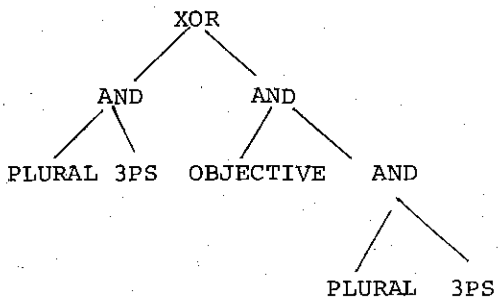
function: nom.obj
predicate: STICK
viewpoint: self
syntactic feature complex:



(iii) FLIES

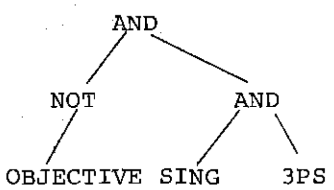
-a-

function: nom.obj
predicate: insect
subpredicate: flying
viewpoint:self
synt.feat.complex



-b-

function: verb
predicate: move
subpredicate: through-air
viewpoint: agent
synt.feat.complex :

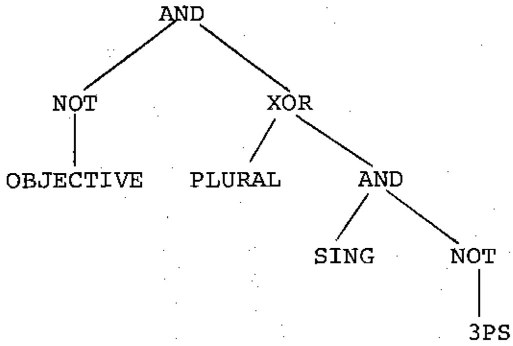


internal feature complex: PRESENT

(iv) LIKE

-a- function:nom.att.adj or nom.adv.adj
predicate: similar
viewpoint: what

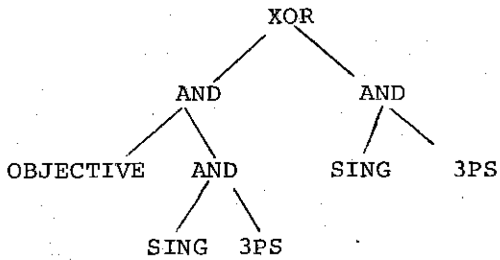
-b- function: verb
predicate: enjoy
viewpoint: agent
external feature complex



internal feature complex: PRESENT

(v) TIME

-a- function: nom.obj
predic:timeline
viewpoint: self
synt.feat.complex



-b- function: verb
predic: measure
viewpoint: agent
ext.feat.complex (AND SING 2PS)
int. feature complex: imperative

We now start a discussion of the parsingprocess. We try to keep the presentation as understandable as possible and avoid formal representations.

Before the first word is consumed the state space should be considered completely empty. Each time a word comes in particles are created and confronted with already existing ones. For ease of reference we number particles according to their moment of creation. For each particle the configuration contained in it will be give explicitly.

INPUTPULSE NR. 1 : TIME

<u>I. Initial particles</u>

The first particles are created for each possible function of TIME according to the lexicon:

(i) <u>Particle 1</u> (for function nom.obj) has configuration

```
            (TIME
                (INP1                    = hypothesis number
                 NOM.OBJ                 = function
                 NIL                     = state in right-going synt.net
                 NIL                     = state in sem. netw
                 ((SING 3PS)(OBJECTIVE SING 3PS)) = synt.feature complex
                 ((THING)(PROPERTY))
                 NIL) )
```

Notice that all information to construct this configuration comes from the linguistic description system. E.g. the semantic features are computed by taking the extension of the features associated with the case frame of TIMELINE (the predicate of time) with the self-case (the viewpoint of time).

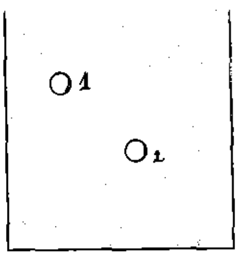
(ii) <u>Particle 2</u> (for function verb) has configuration:

```
    (TIME (INP2 VERB NIL NIL ((PRESENT)) UNDET ))
```

<u>II. Merging</u>

As no other particles are in the state space, nothing more happens and we get as first result:

state space



INPUTPULSE NR. 2. FLIES

I. Initial particles

Again we make a new particle for each function:

(i) particle 3 (for flies as nom.obj) has configuration
(FLIES (INP3 NOM.OBJ NIL NIL ((OBJECTIVE PLURAL 3PS)(PLURAL 3PS))
((ANIMATE)) NIL))

(ii) particle 4 (for flies as predicate) has configuration
(FLIES (INP4 VERB NIL NIL ((PRESENT)) UNDET))

II. Merging of the particles

For each particle of inputpulse 1 and for each particle due to inputpulse 2 it is investigated whether they can merge either from right to left or from left to right. The last one created is always the first one to be investigated further, so we start with investigating particle 4:

Investigate particle 4 (with flies as verb).

1. Let us try to merge this particle with particle 1 embodying INP1 (time as nom.obj)

In other words we investigate whether a nom.obj and a verb may form a link.
From left to right will not do. Although a verb takes objects they come after it, so "time" is in a wrong position to be an object of flies.

From right to left however is a good combination:because
- function-of-head (verb) = nom.obj and time has the function nom.obj. So the function-of-head test is successful.
- position(verb) = after and flies comes after time, hence there is a successful order test.
- The syntactic features match is necessary (a verb agrees with its subject) and it yields true because the features of "flies" are (AND(NOT OBJECTIVE) (AND SING 3PS)) and those of time are ((SING 3PS) (OBJECTIVE SING 3PS)). Notice that the possibility of time having the case signal objective is ruled out.

- The semantic features match yields also true because the viewpoint of flies is agent, the predicate is MOVE and the feature associated is the abstract case frame of MOVE with agent is (XOR ANIMATE THING). Recall that the sema ic features of time in particle 1 are ((PROPERTY)(THING)) . So there is a feature match for the subset ((THING)) as well for modifying as for qualifying.

On the basis of these results it is decided that the particles should merge to form a new one:

particle 5 with the following configuration

```
(FLIES (INP4 VERB NIL NIL ((PRESENT)) UNDET )
  (TIME (INP1 NOM.OBJ NIL NIL ((SING 3PS)) ((THING)) NIL )))
```

Notice that the semantic feature complex of 'time' has been restricted to time as a thing.
Notice also that the predicate forms the top of the structure. This in contrast with the normal procedure of merging particles.

3. We try to merge particle 4 with particle 2 containing INP2 (time as verb).

From left to right will not do with the verb flies because a verb has no head and certainly not a predicate.
From right to left is for the same reason not a good combination. Function-of-head(verb) is nom.obj and nom.adj is not a nom.obj.

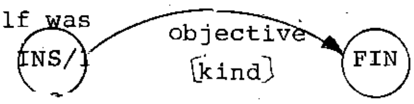
As we now confronted all particles of inputpulse 1 with the particle 4 of inputpulse 2 we can turn to the next particle of inputpulse 2:

(b) Investigate particle 3 (with flies as nom.obj)

(1) We try to merge with particle 1 (time as nom.obj)

example

From left to right the order test is successful because we specified in the grammar that objects may come as well before as after a nom.obj (not necessarily a good assumption in general). Now we investigate the networks. As initial state with flies we have INS/1 . The network itself was



So we go from the initial state INS/1 to the state FIN. The associated case is KIND.
The next step is the matching of the semantic features. This yields also true, because with the KIND-case in INSECT, we have the feature 'property', and property is in the feature complex of time.
We conclude that time is a nom.obj of flies. Notice that this could only be concluded after considering time as some kind of property.

A new particle (<u>particle 6</u>) can now be created:

```
(FLIES (INP3  NOM.OBJ NIL FIN ((OBJECTIVE PLURAL 3PS))
                  (PLURAL 3PS)) ((ANIMATE )) NIL)
    (TIME (INP1 NOM.OBJ FIN NIL ((OBJECTIVE SING 3PS))
                  ((PROPERTY)) KIND) )
```

Notice how the features of the subordinate are restricted and how the case 'kind' has been added, the case state of flies is now FIN.

From right to left a merging is possible according to the position and taking objects tests, however there is no prefix state in the case network of TIMELINE, so we abandon the idea of merging in this direction.

(2) For particle 2 with INP3 (time as verb)

From left to right no merging will take place due to wrong positions.
From right to left we have more success. A verb takes objects and they come after the word, so we proceed with the investigation of what case is filled by 'flies'.

For this purpose we call the semantic network of MEASURE which is the predicate of time, and try to make a transition from the initial state MEAS/1 on the basis of the syntactic feature complex ((OBJECTIVE PLURAL 3PS)(PLURAL 3PS)). The transition is successful and we come in the final state FIN with associated case 'WHAT'. The syntactic features are now restricted to ((OBJECTIVE PLURAL 3PS)). Next we investigate the semantic features. The what case requires (OR THING ANIMATE) and this matches with the feature complex of flies. Hence we may merge the two particles which yields:

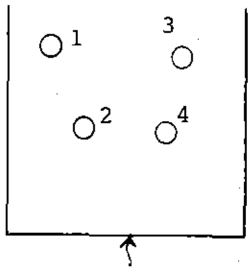
<u>particle 7</u>

```
(TIME (INP2 VERB NIL  ((PRESENT)) UNDET ))
     (FLIES (INP3 NOM.OBJ NIL NIL
             ((OBJECTIVE PLURAL 3PS)) ((ANIMATE)) WHAT))
```

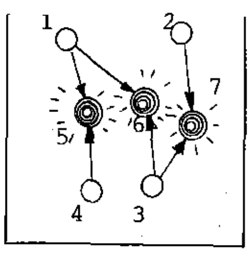
We have now checked all particles of inputpulse 1 against those of inputpulse 2 and obtained some new particles.

Summary of actions in the state space:

from

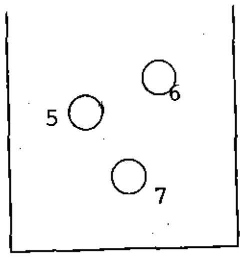


to



Although particle 1,2,3,4 remain in the state space 5,6,7 will be the stronger ones.

So a better representation of the state space at the moment would be:



INPUTPULSE 3 LIKE

I. Creation of new particles

First four initial particles are created for each function assigned to 'like' by the lexicon.

particle 8 with configuration:

(LIKE (INP5 CASESI NIL NIL NIL))

particle 9 with configuration:
(LIKE (INP6 NOM.ATT.ADJ A/1 NIL UNDET))

particle 10 with configuration:
(LIKE (INP7 NOM.ADV.ADJ A/1 NIL ((PRESENT)) UNDET))

particle 11 with configuration
(LIKE (INP8 PREDIC NIL NIL ((PRESENT)) UNDET))

(Notice that in particle 9 and 10 like does not have a final state)

II. Merging

Again we start with the latest made particle to see whether combinations are possible with previously made particles.

(A) Particle 11 with INP8 (like as verb)

1. Let us confront this particle with particle 7 (time as verb)

Neither from left to right nor from right to left is linking possible. A verb does not relate to a verb and vice-versa.

2. Let us confront particle 11 with particle 6 (with flies as nom.obj and time as nom.obj depending from it)

From left to right no merging will take place because the objects of a verb come after their head and not before it. From right to left a merging is indeed possible on the following grounds:
- the head of a verb, i.e. its subject,comes before it, this is the case, hence the test on order is true,
- a verb agrees with its subject, so we have to perform a syntactic features match between (AND (NOT OBJECTIVE) (XOR PLURAL (AND SING (NOT 3PS))) being the features of the verb and ((OBJECTIVE PLURAL 3PS)(PLURAL 3PS)) which is the extension of the features of flies. The match process returns true for the domain ((PLURAL 3PS)). Next we investigate the semantic features via the viewpoint of like (agent) we find that the features of the slot should be ANIMATE; because flies has ((ANIMATE)) this test is again successful and we decide to merge both particles yielding:

particle 12 with configuration:

```
(LIKE (INP8 VERB NIL NIL        ((PRESENT)) UNDET)
   (FLIES (INP3 NOM.OBJ FIN NIL ((PLURAL 3PS)) ((ANIMATE))  NIL)
         (TIME (INP1 NOM.OBJ NIL NIL ((OBJECTIVE SING)) ((PROPERTY)) KIND)))
```

3. Let us finally confront particle 11 with particle 5 (INP3 flies as verb on top)

Both from left to right and from right to left no success is obtained because a verb does not link with another one. Notice that if the verb would have been placed structurally under its head, the merging would in principle be considered but the syntactic feature matches would have resulted in false.

(B) Particle 10 with like as nom.adv.adj

1. Particle 10 in relation to particle 7 (with time as verb on top)

From left to right no merging takes place because the position tests are unsuccessful.
From right to left for the word TIME we have more success.
- The head of a nom.adv.adj is a verb and because flies acts here as a verb, this test is successful.
- Moreover the position of a nom.adv.adj is after its head and this is so.
- There is no synt.features match but there is a sem.feat test. The features associated with the viewpoint of like (which is BETWEEN) are (XOR ANIMATE (XOR THING ACT)). In the frame of MOVE the feature act is associated with the SELF-case (nom.adv.adj is a modifier). Hence there is a match.

The new particle (particle 13) has configuration:

```
 (TIME (INP2 VERB NIL  NIL ((PRESENT)) UNDET)
     (FLIES (INP3 NOM.OBJ FIN NIL ((OBJECTIVE PLURAL 3PS))
                                ((ANIMATE )) WHAT)
         (LIKE (INP7 NOM.ADV.ADJ A/1 NIL MOD) ) )
```

(Notice that like is not in a final state yet)

(2) Let us confront particle 10 with particle 6

From left to right no test is successful , the objects of a nom.adv.adj come after it and not before.
From right to left is not possible because the head of a nom.adv.adj is another adjunct and not an object.

(3) Finally we confront particle 10 with particle 5 (flies as verb on top)

From left to right no success is obtained. The head of flies is an object and not an adjunct. From right to left we are successful:

- The head of a nom.adv.adj is a verb and because flies is a verb, this test is successful;
- Moreover the position of a nom.adv.adj is after its head and this is so;
- There is no syntactic features match, but there is a semantic features test: The features associated with the viewpoint of LIKE (which is WHAT) are (XOR ANIMATE (XOR THING ACT)) . In the features of MOVE we have with the SELF-case (note that nom.adv.adj is a modifier) the feature ACT. So this test is true.

To conclude, we construct the new particle , particle 14, with configuration:

```
(FLIES (INP4 VERB NIL NIL ((PRESENT)) UNDET )
    (TIME (INP1 NOM.OBJ NIL NIL ((SING 3PS )) ((THING)) NIL))
        (LIKE (INP7 NOM.ADV.ADJ A/1 NIL MOD)) )
```

(C) We try to expand particle 9 (with like as nom.att.adj)

Again we confront this particle with all particles active before the inputpulse of like came in.

(1) Confrontation with particle 7.

From left to right will not do. The objects of a nom.att.adj come after their head. Now from left to right.
We start by investigating the word flies. Here we are successful:

- The head of a nom.att.adj is a nom.obj and this is the case;
- The position is as expected;
- There is no syntactic features test, but there is a semantic features test. We have to see whether 'flies' fills a slot in the frame of like, namely the viewpoint of like which is what. To do so the features (XOR ANIMATE (XOR THING ACT)) must be satisfied. This is the case and we get a new particle: particle 15

particle 15 with configuration:

```
(TIME (INP2 vERB NIL NIL ((PRESENT)) UNDET )
  (FLIES (INP3 NOM.OBJ NIL NIL ((OBJECTIVE PLURAL 3PS))
             ((ANIMATE)) WHAT)
      (LIKE (INP6 NOM.ATT.ADJ A/1 NIL UNDET ))))
```

or the word time in particle 7 there is no successful function-of-head test.

(2) Confrontation with particle 6

From left to right no merging will take place because the object of a nom.att.adj should come after 'like'; from right to left we are successful because:

- The head of a nom.att.adjunct is a nom.obj and flies is a nom.obj.
- Moreover the nom.att.adjunct comes after its head and this requirement is fulfilled .
- No syntactic features match is necessary here, but we have a semantic feature match with flies which has the feature ((ANIMATE)). Because the viewpoint of like is between, the features to be satisfied are (XOR ANIMATE (XOR THING ACT)) So the test is successful.

We make a new particle:

particle 16 with configuration:

```
(FLIES (INP3 NOM.OBJ FIN NIL ((OBJECTIVE PLURAL 3PS) (PLURAL 3PS))
             ((ANIMATE)) NIL)
   (TIME (INP1 NOM.OBJ FIN NIL ((OBJECTIVE SING 3PS)) ((PROPERTY)) KIND)
       (LIKE (INP6 NOM.ATT.ADJ A/1 NIL UNDET))))
```

(3) Confrontation with particle 5 (flies as verb on top)

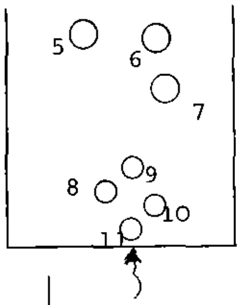
From left to right and from right to left no success is obtained due to the function-of-head tests. A nom.att.adj has as head a nom.obj and not a predicate whereas the head of a predicate is a nom.obj and not a nom.att.adj.

(D) Particle 9 (with INP5, like as case sign)

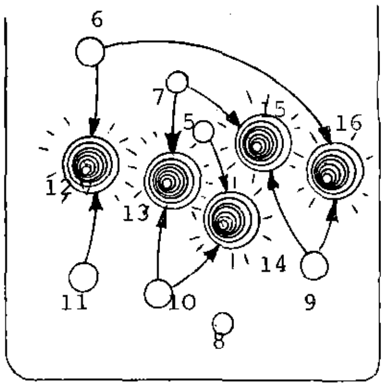
All confrontations with previous particles yield false as the reader can find out for himself. The cause is always the function-of-head test.

The particles resulting from the third input pulse 'like' have caused a strong activity in the state space.

In particular we went from:



to



We will carry on with the most powerful particles in the state space.

INPUTPULSE 4 AN

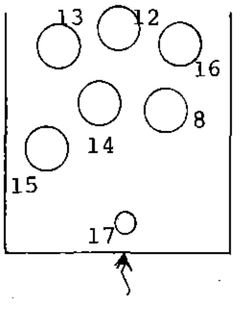
I. New particles

There is only one: partilce 17 with configuration

(AN (INP9 DET FIN))

II. Merging

For all particles the tests will be unsuccessful. On the basis of the function-of-head tests and/or order tests, so we are left with the following state space:



INPUTPULSE 5 : ARROW

I. New particles

There is again only one particle: particle 18.

(ARROW (INP10 NOM.OBJ NIL NIL
((OBJECTIVE SING 3PS)(SING 3PS)) ((THING)) NIL))

example

II. Merging

(A) We try to merge   particle 18

(1) With particle 17

Due to one of our principles that you cannot 'hop' over a word, the first job is to merge with particle 17. This is possible from left to right because:

- A determiner makes a transition from the initial state (OBJ/1) associated with the nom.obj 'arrow' which brings us in the network in the state OBJ/2 ;
- moreover the syntactic features match is successful, 'AN' has 'SING' and arrow has ((OBJECTIVE SING)) So there is a match. Also we have to send-through the feature 'UNDEF' which brings us to the new feature complex ((OBJECTIVE SING UNDEF)). No more tests are necessary which brings us to the new particle:

<u>particle 19</u> with configuration

```
(ARROW (INP10 NOM.OBJ NIL NIL  ((OBJECTIVE SING UNDEF)(
                 SING UNDEF)) ((THING)) NIL)
    (AN (INP9 DET NIL)) )
```

We now have the opportunity to show what happens if a particle is made and it does not cover the whole input sentence yet. In such a situation a chain reaction can be said to take place: We try to merge with other particles floating aroung on the border of the range of this particle. The whole process is set in motion again by placing particle 19 on the takslist which is a pushdownstore; this implies that it is the first particle again considered for further combination.

(B) We try to expand particle 19

(1) Let us confront it with particle 8 (like as casesign)

Recall that the latest state associated with nom.obj was OBJ/2 .So we try to make a transition in the network which brings us to the new state OBJ/3. Although there is no syntactic feature match, we have to pass features to the feature complex of the head.

This yields particle 20 with configuration

```
    (ARROW (INP10 NOM.OBJ NIL NIL ((3PS SING OBJECTIVE UNDEF LIKE))
                    ((THING)) NIL)
        (AN (INP9 DET NIL)) (LIKE(INP5 CASESI NIL)) )
```

Notice how the case sign is now in the feature complex of the nom.obj and ready to become active in surface case signal tests. To show this was the reason to incorporate 'like' in this function. No further results with this particle will be obtained.
From right to left there is no merging possible because 'like' (as casesign) takes no objects.

(2) Let us confront particle 19 with particle 16 ('flies' as nom.obj on top)

From left to right the order test and the taking-objects test is true. But we did not include a semantic network for 'flies' and therefore do not investigate the possibility any further.

From right to left we are successful for the word like . Like is a nom.att.adj it takes objects and they come after it. The transition in the sem.netw is also successful. We go from the state SIMIL/1 to the new state FIN with associated case TO for the syntactic feature complex ((3PS OBJECTIVE SING UNDEF)). The sem.features test yields also true and we get a new particle:

particle 21 with configuration:

```
(FLIES (INP3 NOM.OBJ NIL NIL ((OBJECTIVE PLURAL 3PS))
                                    ((THING )) NIL)
    (TIME (INP1 NOM.OBJ NIL NIL ((OBJECTIVE SING 3PS)) ((PROPERTY)) KIND)
          (LIKE (INP6 NOM.ATT.ADJ FIN NIL  UNDET)
               (ARROW  (INP10 NOM.OBJ NIL NIL ((3PS SING OBJECTIVE UNDEF))
                    ((THING)) TO) (AN (INP9 DET NIL)))))
```

Notice how 'like' has entered a final state and how the case has been added.

particle 21 is the first particle which is final in the sense that it covers the whole input sentence .

From right to left no further combinations are possible for the word flies (no transition in sem.netw).

(2 .2) For particle 15

From left to right will not do because a verb comes after the object which is its subject. From right to left there is greater success. Take the word like (function nom.att.adj) It is obvious that on the same basis as for the creation of particle 21 we will be able to link the object to like. Hence we get a new particle:

particle 22 which is again final:

```
(TIME (INP2 VERB NIL NIL ((PRESENT)) UNDET)
       (FLIES (INP3 NOM.OBJ NIL NIL ((OBJECTIVE PLURAL 3PS))
                    ((ANIMATE )) WHAT  )
            (LIKE (INP6 NOM.ATT.ADJ NIL NIL UNDET)
               (ARROW (INP10 NOM.OBJ NIL NIL ((3PS OBJECTIVE SING UNDEF))
                             ((THING)) TO)
                   (AN (INP9 DET FIN))))))
```

Still from right to left for the word flies, no linking takes place because there is no transition possible. For the same reason we cannot merge for the word time.

(3) For particle 13.

From left to right no merging takes place because a verb (which is on top of 13) stands after its subject. However from right to left we are again successful. This time for the word 'like'. Again on the same basis as for the two previous particles.

The new particle (particle 23) has configuration

```
(FLIES (INP4 VERB NIL NIL ((PRESENT)) UNDET)
    (TIME (INP1 NOM.OBJ NIL NIL ((SING 3PS)) ((THING))    )
      (LIKE (INP7 NOM.ADV.ADJ FIN NIL MOD)
         (ARROW (INP10 NOM.OBJ NIL NIL ((3PS OBJECTIVE SING UNDEF))
                          ((THING )) TO)
               (AND (INP9 DET NIL )))))
```

Still from right to left we try to merge for the word 'flies'. This does not work because no transition is possible in the semantic network.

(3.2.) Particle 14.

From left to right will not do because a verb comes after its subject.
From right to left is more successful. Not for the word flies because no transition is possible in the sem. network .
But for the word like, the order test is successful and there is a transition from SIMIL/1 to the new state FIN. The sem.feat test is also successful which leads to a new particle: particle 24 with configuration:

```
(TIME (INP2 VERB NIL NIL ((PRESENT)) UNDET )
     (FLIES (INP3 NOM.OBJ NIL NIL ((OBJECTIVE PLURAL 3PS))
                   ((ANIMATE)) WHAT))
        (LIKE (INP7  NOM.ADV.ADJ  NIL NIL MOD)
           (ARROW (INP10 NOM.OBJ FIN NIL ((3PS OBJECTIVE SING UNDEF))
                        ((THING)) TO) (AN (INP9 DET FIN)))))
```

For the word time there is no transition in the semantic network although the ordertest was successful.

(4) For particle 12

Here we are successful from right to left (from left to right is not investigated because the top is a verb) . First of all the order test and taking objects test are successful for like, also we can perform a transition in the case frame of ENJOY and the semantic features test is successful. This leads to the following new particle: particle 25:

with configuration

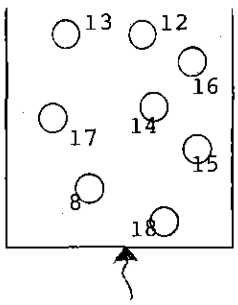
```
(LIKE (INP8 VERB NIL NIL  ((PRESENT)) UNDET)
   (FLIES (INP3 NOM.OBJ NIL NIL ((PLURAL 3PS)) ((ANIMATE)) NIL)
       (TIME (INP1 NOM.OBJ NIL NIL ((OBJECTIVE SING 3PS))
                     ((PROPERTY)) KIND)))
       (ARROW (INP10 NOM.OBJ FIN NIL ((3PS OBJECTIVE SING UNDEF))
             ((THING )) WHAT ) (AN (INP9 DET FIN ))))
```

(C) It remains to be investigated how particle 19 can be further expanded.

The investigation of this is left to the reader. There will be no successful mergings.

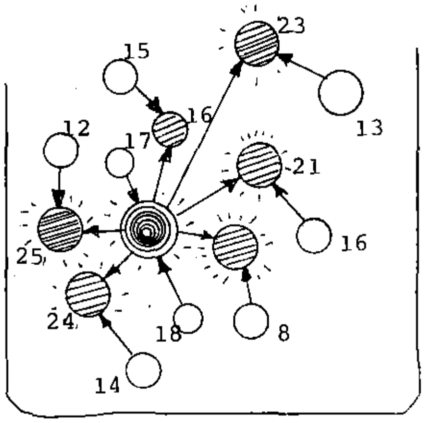
As a summary of actions due to this inputpulse we get:

from

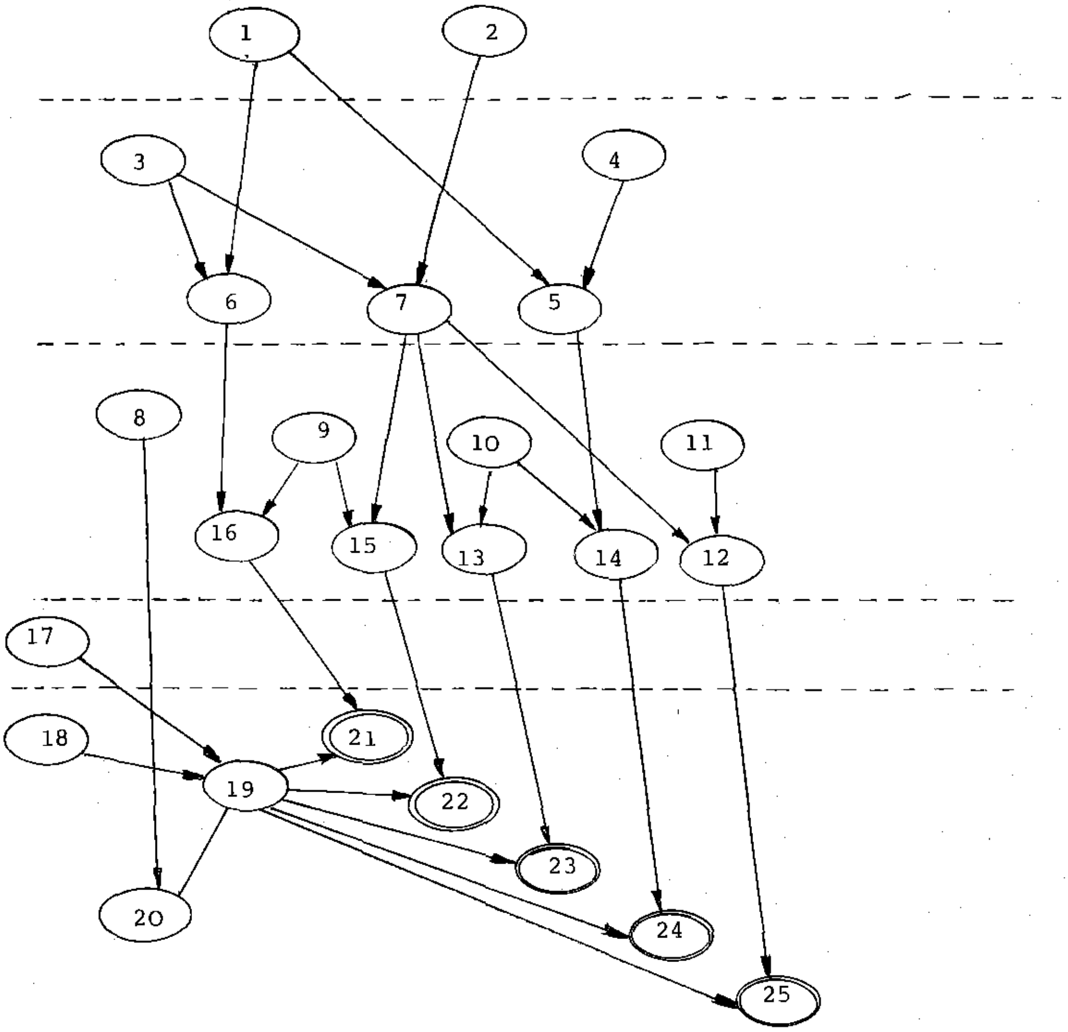


example

to



Here is a summary of all actions on particles that occurred during the analysis of the sentence:



(Final particles have double rings)

2.1.6. The computation of the resulting structures

We now discuss how it is possible to extract from a particle the structures defined earlier. These structures (even the semantic ones) are all auxiliary constructs mainly used for didactic purposes. In principle semantic interpretation can take place immediately on the basis of the information contained in a particle. (Notice how the distinction deep/surface structure disappears).

(i) The functional structure

It is possible to extract a functional structure (as defined earlier) from the configuration in a particle by means of the function F-struct:

<u>Definition</u>

Let $a_k = \langle a_{1,k}, a_{2,k}, a_{2+1,k}, \ldots, a_{2+j,k} \rangle \qquad j \geqslant 0$

be a configuration with

$a_{2,k} = \langle i_{1,k}, i_{2,k}, \ldots \rangle$ an information sequence

then

$$\text{F-struct}(a_k) = \begin{cases} (i_{2,k} \;\; a_{1,k}) & \text{for } j = 0 \\ (i_{2,k} \;\; (a_{1,k} \; \text{F-struct}\,(a_{2+1,k}) \ldots \text{F-struct}(a_{2+j},k)\;) & \text{for } j > 0 \end{cases}$$

Notice that this yields a list structure which is coverted into a tree by the standard conventions.

(ii) The case structure

It is possible to extract case structures from a particle by means of the following method:

Definition

Let $a_k = \langle a_{1,k}, a_{2,k}, a_{2+1,k}, \ldots, a_{2+j,k} \rangle \quad j \geqslant 0$

be a configuration
with
$a_{2,k} = \langle i_{1,k}, i_{2,k}, \ldots \rangle$ an information
sequence
then

(i) $\langle a_{1,a_{2+i,k}}, a_{1,k} \rangle \in$ case structure

with

$$\text{label}\ (\langle a_{1,a_{2+i,k}}, a_{1,k} \rangle) = i_{7,a_{2+i,k}}$$

iff $i_{2,a_{2+i,k}} \in$ F-obj for $1 \leqslant i \leqslant j$

and

(ii) $\langle a_{1,k}, a_{1,a_{2+i,k}} \rangle \in$ case structure

with

$$\text{label}\ (\langle a_{1,k}, a_{1,a_{2+i,k}} \rangle) = i_{6,k}$$

iff $i_{2,k} \in$ F-adju $\quad 1 \leqslant i \leqslant j$

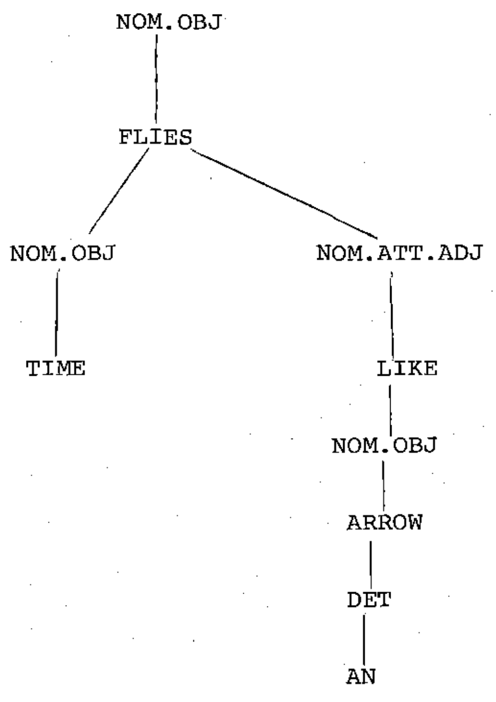
Some examples

We give some particles of the earlier discussed example of the parsing process and present each time the functional and case structure.

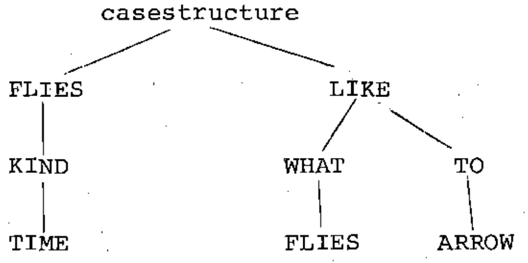
For particle 21 with configuration:

```
(FLIES (INP3 NOM.OBJ NIL NIL ((OBJECTIVE PLURAL 3PS)) ((THING)) )
    (TIME (INP1 NOM.OBJ NIL NIL ((OBJECTIVE SING 3PS)) ((PROPERTY))
                                                  KIND)
     (LIKE (INP6 NOM.ATT.ADJ FIN NIL UNDET)
        (ARROW (INP10 NOM.OBJ FIN NIL ((3PS OBJECTIVE SING UNDEF))
                     ((THING)) TO)
            (AN (INP9 DET NIL)) ) ))
```

functional structure



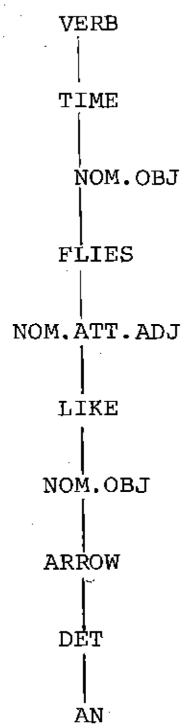
case structure:



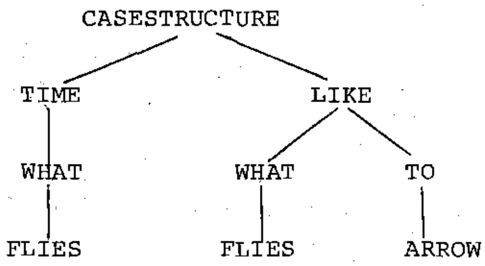
For particle 22 with configuration:

```
(TIME (INP2 VERB NIL NIL UNDET ((PRESENT)) UNDET )
     (FLIES (INP3 NOM.OBJ NIL NIL ((oBJECTIVE PLURAL 3PS ))
                 ((ANIMATE)) WHAT)
       (LIKE (INP6 NOM.ATT.ADJ NIL NIL UNDET ) )
          (ARROW (INP10 NOM.OBJ FIN NIL ((3PS OBJECTIVE SING UNDEF))
                       ((THING)) TO)
              (AN (INP9 DET NIL) ) ))))
```

functional structure:



case structure:



(iii) Semantic structures

The extraction of the semantic structures in the format of the SRL language is a straighforward process. It works on the basis of a task oriented control structure just as the parser itself.

A task here contains two things (i) a pointer in the structure of the particles, (ii) an attachment point, i.e. a point where the structure resulting from executing the task should be attached in the already obtained semantic structure. This attachment point is in fact a set: a point for if the function of the word in the configuration addressed to by the pointer is of type object, then the attachment point is the list of cases in the head of the object, a point for if the function is of type qualifying adjunct, then the attachment point is the variable node of its head and a point for if the function is of type modifying adjunct, then the point is the predicate structure of its head.

The initial task contains a pointer to the top of the structure; the attachment points are NIL.

The system takes each time the algorithm on top of the tasklist. Then the task is executed according to the following specifications:

If the word on top of the configuration pointed at in the task is of type object

(i) create a new object node

(ii) hang the viewpoint, predicate and subpredicate as specified in the lexicon under the predicate node

(iii) add features if any

(iv) construct a new task for all depending nodes

(v) if the object fills a slot in a case frame, attach the case label and the pointer to the object node in the semantic structure under the node defined in the attachment point.

If the word on top of the configuration pointed at in the task is of type adjunct

(i) make a viewpoint/predicate/subpredicate frame and hang it under the attachment point indicated in the task

(ii) add features if any

(iii) construct new tasks for all depending nodes.

If the word on top of the configuration pointed at in the task is of type functionword

(i) construct new tasks for all depending nodes.

Extensive examples and detailed descriptions of several semantic structuring processes will be given in the chapter on examples and experimental results.

Notice how the distinction between objects/adjuncts/functionwords which proved to be basic for the formulation of the grammar rules is also fundamental to the semantic structuring process as we have predicted.

2.2. The PRODUCTION PROCESS

In this section we present a short outline of the production process based on the modular grammar theory. We will not present a very detailed model for two reasons (i) the size of the present work would grow out of the envisaged proportions, (ii) the deadline forced us to remain in the presentation here on a rather intuitive level. This does not mean however that the investigation on the production process was not carried out within our general methodological framework (i.e. that computer programs should be constructed to prove the operational capacities of the approach). In fact we worked extensively on a system for producing natural language even before starting out for the parsing problem (results are reported in Steels, 1976); and many important discoveries were made during the investigation of language production rather than recognition.

In particular the idea that grammatical function is one of the basic factors in language functioning (more basic than grammatical cateogry) and the idea of 'viewpoint' as a way to compute surface case frames from abstract case frames and thus to provide an alternative for transformational grammars on this point were both discovered during studies in language production.

By the production of natural language, we do not mean the generation of a sentence from an initial symbol by successively applying the derivation relation on the basis of some generative grammar, but rather the realization of a mapping from information contained in a store into sentences of some natural language.

Although recent work in transformational grammar is more and more approaching the same subject matter, it must be noted that there is a fundamental distinction between generating and producing.

Generation is a process precisely defined in the theory of formal grammars as an operation over strings (called a derivation) which when applied in sequence as controlled by the rules of the grammar results in one sentence of the language that is to be defined. One of the main features of this concept of generating is that it is uncontrolled, that means if somewhere in the grammar two paths are possible there is no mechanism that tells what path should be followed.

Production is a transduction process and it is assumed that every action that is undertaken finds its final motivation in the intenstion of the system. In other words a producing system is a goal-directed system, it wants to convey information and uses certain means for that. It follows that to construct a successful producing system we must represent in the grammar the relation between a certain intension and how this intension is made clear to the reader/listener according to conventions agreed upon.

We claim that the modular grammar that was introduced previously contains just the kind of knowledge we will need in order to produce natural language. Even more, while we needed for the parsing process special predicates (the parsing predicates) it turns out that we now can consult the knowledge directly. So, if a modular grammar is biased, it would be as regards production (and not as regards analysis as probably all readers have been thinking).

Intuitive explanations of the model.

Let us again start from the 'particle concept' as used to explic ate the parsing process. Now the particles will be called tasks because that seems an easier way to capture the ideas we have in mind. There are two sorts of tasks, the first type contains the basic impulse to create language code for a certain piece of semantic information (we call this a taskbuilder task). This task then enters the language production space and is expanded to a sequence of other tasks. The new tasks are of two sorts, either from the first type agin,

i.e. a request for new impulses from the semantic processes,
from a second type, the so called lexicalisation tasks. A lexicalisation task contains every information that is necessary to produce one single word. It is handed over to the dictionary routines which produce then the word itself .

The crucial point in the system is of course the moment of taskbuilding. This involves two aspects (i) the scheduling of the tasks and (ii) the determination of what information should be put in a newly formed task. It is performed on the basis of the various knowledge sources already discussed. Each module (or in other words each specialist for a particular part of the language) is asked to contribute in order to accomplish the complex job.

From the explanations it follows that the following points need to be clarified (i) the exact definition of the contents of the tasks, (ii) the control structure for the execution of the tasks and (iii) the process of executing a task.

2.2.1. The tasks

There are two sorts of tasks:

(i) Taskbuilder tasks which contain a pointer to a node in the semantic structure that is to be recoded in a natural language. These tasks consitute the 'stimuli' for the production system to become active.

Definition

A taskbuilder task is a 4-tuple ⟨a1,a2,a3,a4⟩
with

a1 = the keyword TKB (taskbuilding)

a2 = a pointer to the task which was the immmediate source for this task

a3 = a pointer to a node in the semantic structure

a4 = a feature complex which is already due to earlier processing.

(ii) Lexicalisation tasks which contain all necessary information for the dictionary lookup process to do its job.

Definition

A lexicalisation task is a 6-tuple ⟨a1,a2,a3,a4,a5,a6⟩
with

a1 = the keyword LEX

a2 = the function of the word

a3 = the predicate

a4 = the subpredicate

a5 = the viewpoint

a6 = the feature complex(es)

No other sorts of intermediate representation constructs will be used. In other words everything else is in the process defined upon the tasks.

2.2.2. The process

Ideally a producing system should be able to reason about language in a similar fashion as the parsing system discussed in previous section did. Such a reasoning process could again be organized in a nondeterministic process by organizing particles which cover a whole sentence. (Cf. hints in this direction when discussing the transduction relation for completion networks).

In the simpler account given here we assume that the process of language production is straight forward and probably the more we learn about language the more it will turn out to be very strongly determined how a sentence should be produced in view of certain meaning, context, situation,etc.

As regards the control structure of the system we need the following:
(i) a store on which tasks are placed in a last in first out manner
(ii) a function which takes one task and sends it either to the taskbuilder (if it is a taskbuilder task) or the dictionary specialist (if it is a lexicalization task). If there are no tasks left the sentence is complete.

Let us now provide some more detail on the taskbuilder and the dictionary specialist.

(a) The taskbuilder

-i- The computation of the factors

The first assumption underlying the operation of the system is that one can compute on the basis of the semantic structures what the grammatical function of a predicate in the structure will be. This is the exact reverse of the semantic structuring process discussed before. There we saw that a particular grammatical function implies a particular sort of semantic structure. Now we reverse this relation: a particular semantic structure implies a particular grammatical function.

Obviously this relation (and its reverse) are strongly depending on the type of grammatical functions that the linguist designing an empirical interpretation for a particular natural language wants to use.

A second assumption is that it is possible to compute the viewpoint. When a TKB-task is resulting from a previous TKB-task this viewpoint is the semantic relation holding between the two nodes in the respective TKB tasks. When the TKB task contains an object (as happens most of the time for the first task) the viewpoint is the relation between the predicate used to introduce the object and the entity node itself.

If there are some more factors introduced in the grammar later on, they should also be computable on before hand.

-ii - The scheduling of the other tasks

Once it is known what the function of the predicate pointed at in the task is, we have acces to the grammar (i.e. to all rules with factor function/case, to the synt. networks, to the case networks , etc;)
The first question the system now asks is what other information in relation to the predicate in the current TKB-task should be communicated.

A list is made of these information tuples and then the list is split in two parts. One containing tasks to be scheduled before the present task and other tasks to be scheduled after it, and each sublist is internally ordered. This scheduling process is performed on the basis of the networks (recall here the transduction relation defined in relation to the completion networks) and the rules on order. Because the respective tasklists (before/after) are used as pushdownstores, we obtain the right paths in the networks.

-iii- sending through information

Although the newly made tasks may be other TKB tasks,normally information is sent through to the new task s in the form of features . (For TKB-tasks in the fourth position). E.g. when going through a case network specification (AND BY OBJECTIVE) may be obtained as side effect of a transition in the network for a particular case (cf.government rules). This feature is sent to the new task introducing that object.

When performing the taskbuilder actions for the task of the object, we will introduce a functionword 'casesign' for the by feature, etc;.

-iv- Lexicalization tasks

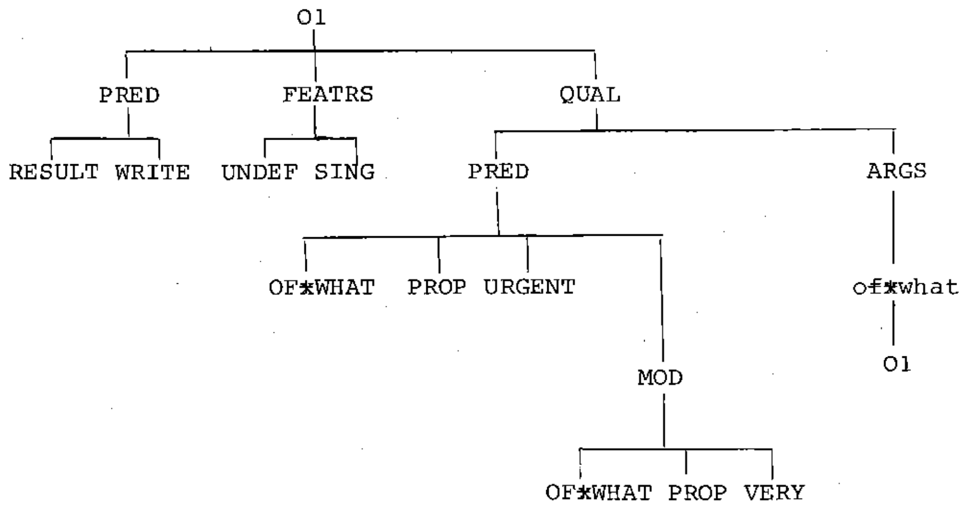
When every job has been performed in relation to the TKB task under investigation by the taskbuilder, this task is turned into a lexicalisation task itself, i.e. all relevant information is grouped according to the format specified. Then all tasks made are placed on the main tasklist and the system starts investigating the first task on top of this list.

(b) The dictionary specialist.

The dictionary specialist scans the dictionary in reverse mode. Earlier we had a word and from this we searched for the information tuples related to this word. Now we go the reverse way. To optimize the process, we have pointers from each (concrete) predicate to all relevant words and further to all subsets of a given function. The rest of the search is performed by the match processes of the feature calculus which work in both directions anyway.

2.2.3. Example

Let us now give a short example of a production process for the example phrase "A very urgent letter" , in other words we realize one piece of the semantic structure in particular:



STEP 1

First we make the initialization task pointing to the O1 node itself:

1. ⟨TKB , Ø, O1, NIL ⟩

STEP 2

The first job in the execution of this task consists in computing the function , the predicate and the viewpoint. The answers are straightforward : function: nom.obj (because we have an entity introduced by a predicate), pred: write, viewpoint: result.

Next we make a list of depending information items: features and qualifiers. For each of these itmes we investigate possible functions, yielding determiner for feature undef and att.adjunct for qualifier with predicate PROP (because it is in adjunct of a nom.obj).

Investigating the networks and the order rules in the grammar we find that a tasklist of items 'before' contains the determiner and the qualifier with predicate urgent.

The next step is to construct a lexicalization task for the nom.obj its f. All these tasks are then put on the tasklist and we get:

3 . ⟨LEX, DETERM, NIL, NIL , NIL , ((UNDEF))⟩
2. ⟨TKB, 2, QUAL, NIL⟩
1. ⟨LEX,NOM.OBJ , WRIT, NIL, RESULT, ((SING))⟩

(Notice that for functionwords the lexicalisation task could be made immediately)

STEP 3

Now we proceed by investigating the first task on the tasklist. This task is a lexicalisation task. So we go into the dictionary and we find there the word 'a'. The remaining tasklist now looks as follows:

2. ⟨TKB , 2, QUAL, NIL⟩
1. ⟨LEX, NOM.OBJ, WRIT, NIL, RESULT,((SING))⟩

STEP 4

The next task is again a taskbuilding task. We make a list of depending terms. This contains one modifier, for predicate PROP, The function of this modifier is adv.adj (modifier of an att.adj). We know from the grammar that an adv.adj comes before its att.adj Hence we put the task to realize the modifier node on the 'before' list. As there are no other items, we construct a lexicalisation task for the predicate in this task. As final result we get:

3. ⟨TKB , 4, MOD, NIL⟩
2. ⟨LEX , ATT.ADJ, PROP, URGENT, OF*WHAT , NIL ⟩
1. ⟨LEX, NOM.OBJ , WRIT, NIL, RESULT, ((SING))⟩

STEP 5

The task on top is a taskbuilder task. We look into the structure but we don't see any depending nodes. Therefore the only thing necessary is to construct a lexicalisation task for the modifier. The function is adv.adj; the predicate PROP and the viewpoint OF*WHAT

Resulting tasklist:

3. ⟨LEX, ADV.ADJ , PROP, VERY, OF*WHAT ⟩
2. ⟨LEX, ATT.ADJ , PROP, URGENT, OF*WHAT ⟩
1. ⟨LEX, NOM.OBJ , WRIT, NIL, RESULT ((SING))⟩

STEP 6

We execute the remaining lexicalisation task which yields as output 'A VERY URGENT LETTER'.

§ 3. THE IMPLEMENTATION

In this chapter we present the details of the computer implementation we have constructed for the parser discussed in the previous chapter. In a first section we introduce a number of auxiliary routines which together constitute a library for list processing in FORTRAN IV. In a second section we come to the implementation of the parser itself.
In a final section we give the routines which compute the functional, case and semantic structure out of final particles as computed by the parser.

§ 3. THE IMPLEMENTATION

3.1. INTRODUCTION TO THE IMPLEMENTATION

The programming language FORTRAN IV will be used here as the formal language for the representation of the algorithms. To computational linguists this may come as a surprise . It is well known that FORTRAN IV is a very 'tough' language for linguistic applications: no list processing, no easy symbol manipulation, no recursive programming. The reason for taking FORTRAN was simply that at the time the investigations started, no other language was available on the PDP 11/45 we are using in our laboratory. Although we later on managed to implement a LISP interpreter system, the working space of this interpreter soon proved to small for the kind of programs we will be discussing.

This restrictedness of memory (32 K)was a second major decision factor in favour of FORTRAN. It is necessary to write highly efficient programs , especially as regards memory requirements, on such a small machine as a PDP 11/45.

The choice (or rather necessity) for using FORTRAN has the advantage that the programs will be understandable by a large group because FORTRAN IV is the most widespread programming language. Also, the programs can be implemented all over the world because FORTRAN is available in practically every computer centre.

The first thing necessary however to be able to use FORTRAN successfully for linguistic applications is the implementation of a number of functions and subroutines which complement FORTRAN with list processing capacity. The discussion of these functions and subroutines is the purpose of this introduction

(1) List processing in FORTRAN IV.

List processing involves a way of representing internally in the machine all the information about lists and about the atoms contained in them. Also we need ways to input and output lists and atoms and to perform operations on lists. The first question we deal with is the representation problem.

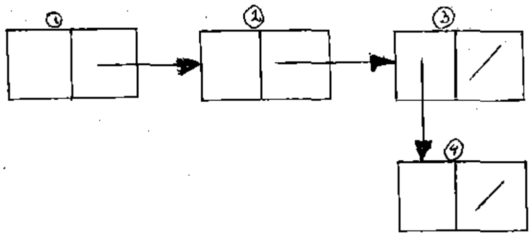
Representation

A list is a number of cells linked on each other by means of pointers. It follows that we need a way to represent the cells and to represent the pointers. A cell contains three parts the atomflag (AF), a place to store the car of the cell (CAR), and a place for the cdr of a cell (CDR).

If we now organize three vectors, respectively called AF, CAR, CDR and let the parameters of the vectors be the address of the cell then we have not only a way to represent a cell I (by a triple AF(I), CAR(I), and CDR(I)) but also a way to point at cells, namely by the parameter: I. In addition we can address each part of the cell seperately.

Example:

The list (A B (C)) is graphically:



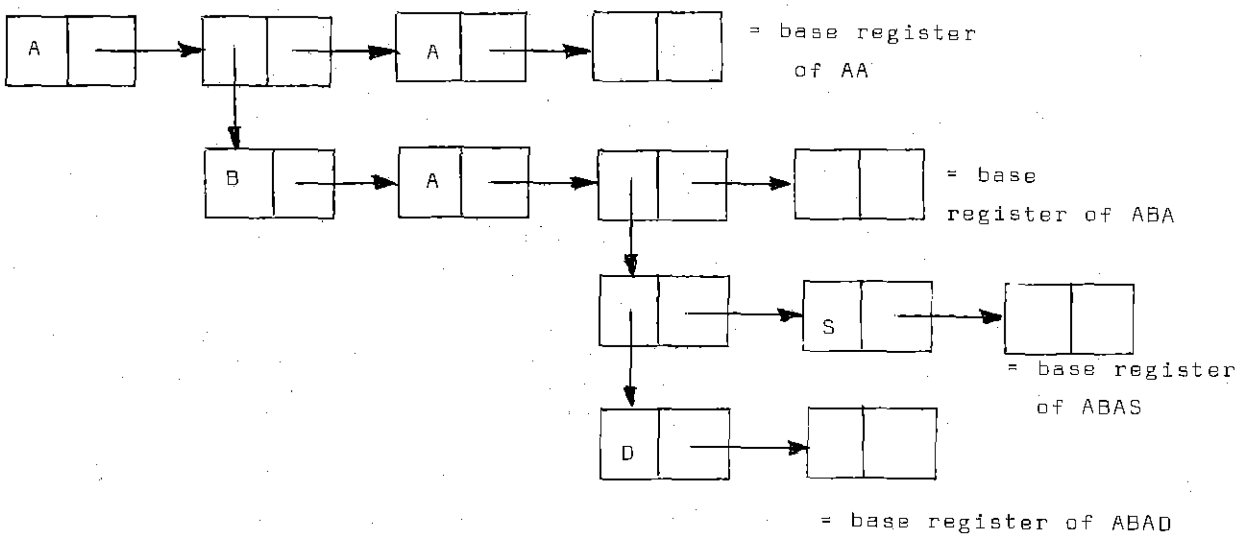
then the FORTRAN representation will be

	AF	CAR	CDR
1.	Ø	A	2
2.	Ø	B	3
3.	Ø	4	Ø
4.	Ø	C	Ø

Note that the representation of NIL (the null list) is Ø.

Now for atoms we need (i) a dictionary in which the atoms are stored, (ii) a base register, i.e. a unique cell that will be used as unique address of the atom and (iii) a property list on which at least the printname is stored.

For the dictionary we will also use a list structure, based on the principle that equal front parts are stored only once. E.g. the atoms AA, ABA, ABAS, ABAD are stored in a structure with in the cars single characters:



Notice that on each end of a path there hangs the base register of the atom made up by the characters of that path. The cells in the dictionary structure and all base registers have 1 in the atomflag (AF) of the cell. All the others have Ø. This is needed to keep both types strictly apart.

The property list is a special list of pairs (property, value) which is stored in a condensed form. The property list hangs on the CDR of the base register of the atom. The first item is always a pointer to the printname of the atom. After that comes a special list of cells where the CAR contains the property and the CDR the value.

So a complete FORTRAN representation (except for the dictionary) for the list (A B (C)) would be

	AF	CAR	CDR	
1.	Ø	5	2	
2.	Ø	6	3	
3.	Ø	4	Ø	
4.	Ø	7	Ø	
5.	1	Ø	8	= base register of A
6.	1	Ø	1Ø	= base register of B
7.	1	Ø	12	= base register of C
8.	Ø	9	Ø	property list of A
9.	A	Ø	Ø	printname of A
10.	Ø	11	Ø	property list of B
11.	B	Ø	Ø	printname of B
12.	Ø	13	Ø	property list of C
13.	C	Ø	Ø	printname of C

In the current implementation we have 3000 cells available. The AF is declared LOGICAL*1 data type and the CAR and CDR as INTEGER*2 . All three vectors are placed in a commonzone.

Note that as a consequence of these options all pointers either to lists or to atoms are of INTEGER*2 data type !

With this representation in mind, we can now turn to the routines which perform the input/output and processing.

Processing

In a list processing system there is normally a so called freelist created at the start. When in need of a piece of list structured memory, one takes 'cells' from this freelist and when these cells are no longer needed, they are returned to the freelist. The creation of this freelist is the task of a special subroutine INIT. After this subroutine is called, the system is ready to start.

The pointer to the freelist is called IFREE and available in a commonzone called /IFREE/.

Next we need a routine for input (RLIST) and one for output (PRLIST). In addition we have a program to plot automatically tree structures on the plotter. PLOTLI is the preparation of this program.

For doing list processing, we have a routine for taking cells from the freelist (NEW) and one for returning them (BACK).

Lists are copied by COPY and erased by ERASE.

A pushdownstore can be simulated by using the routines PUSH and POPUP.

Work on the property list is performed by PROP and GET.

Routines which hang new list structures on already existing ones are ADD, APPEND, and ATTACH.

To check whether we are dealing with a list or an atom, we use the predicate ATOM and LIST.

All routines are grouped together in a library called the FORLI.OLB library.

Before we start a discussion of the routines in detail, we give a detailed example of the operation of one single subroutine. This may help the reader in reading and understanding the other ones. Let us consider the subroutine APPEND (see first its definition on one of the following pages). We consider APPEND in connection with the following main program:

```
1.   IMPLICIT INTEGER (A-W)
2.   LOGICAL*1 AF
3.   COMMON  CAR(3ØØØ),CDR(3ØØØ),AF(3ØØØ)
```

```
4.   I1 = RLIST (1,I,1)
5.   I2 = RLIST (I,I,1)
6.   CALL APPEND (I1,I2,J)
7.   CALL APPEND (J,I2,J)
8.   CALL  PRLIST(I1,1,6)
9.   END
```

What happens in this little program is this. First we read a list from a device with logical unit number 1 (e.g. the card reader) starting with the first character on the card. The list is pointed at by I1.
Then the system reads another list (or an atom) on the same line and sets a pointer I2 to it. By calling two times APPEND we then add the second one two times to the first one.
E.g. if we read I1 = (A) and I2 = B then after the first APPEND we get (A B) and after the second (A B B). The result is printed by PRLIST on a device with logical unit number 6 and from the first item on the next output line.

Now let us trace exactly what happens in APPEND. Given (hypothetically) the following (simplified) FORTRAN representation after RLIST (in line 3) of main program):

	CAR	CDR		
1.	A	Ø	=	I1
2.	Ø	3	=	beginning of freelist
3.	Ø	4		

⋮

Notice that we leave out AF indicators for simplicity.

Now we enter APPEND with I1 = 1, I2 = B and I3 undefined. IFREE = 2.

First we take a new cell from the freelist. CDR(1) becomes 2 (line 6) put I2 in its car: CAR(2) becomes B (line 7), note the provision for exhausting the memory in line 8, I3 = 2 (line 9), IFREE (equal to 2) is advanced to CDR(2) = 3 in line 10 and finally CDR(2) = Ø. This yields:

	CAR	CDR	
1.	A	2	= I1
2.	B	O	= J,I3
3.	O	4	freelist
4.	O	5	

⋮

Then we enter APPEND again with I1 = 2, I2 = B, I3 yet irrelevant and IFREE (in the commonzone) is 3.

First we take a new cell from the freelist CDR(2) = 3 (line 6), put I2 in the CAR(3) = B (line 7), set I3 equal to the new cell I3 = 3 and advance IFREE = CDR(3) = 4.
Finally CDR(3) = O.
This yields:

	CAR	CDR	
1.	A	2	I1
2.	B	3	
3.	B	Ø	
4.	Ø	5	= freelist
5.	Ø	6	

⋮

From this example it should be obvious what complicated list processing activities are going on in the computer when we come to serious programs such as a parsing system for example. To trace the analysis of one sentence in the detail just provided is an almost impossible thing to do.

Now we discuss the routines that make up the library and thus form the groundwork for the further implementations. The routines are appearing in alphabetic order.

ADD

parameters: I2, I1.
I1 is a list and I2 is an atom or a linear list of atoms.

operation: After execution of ADD, each atom of I2 is added to I1 if and only if it is not present yet.

example: Let I2 = (C B A) and I1 be (A B C) then after CALL ADD(I2,I1) I1 will be (A B C).
Let I1 = (A B C) and I2 = (D E F) then after CALL ADD (I2, I1) I1 will be (A B C D E F)

code:

```
      SUBROUTINE ADD (I2,I1)
      IMPLICIT INTEGER (A-W)
      LOGICAL*1 AF
      COMMON CAR(3000),CDR(3000),AF(3000)
      NIL = 0
      IF(I2.EQ.0) RETURN
      FLAG = 0
      IF(AF(I2).NE.1) GOTO 1
      L = I2
    5 J = I1
    2 IF(CAR(J).EQ.L) GOTO 4
      IF(CDR(J).EQ.0) GOTO 3
      J = CDR(J)
      GOTO 2
    3 CALL NEW(I)
      CDR(J) = I
      CAR(I) = L
      GOTO 4
    1 FLAG = 1
      K = I2
      L = CAR(K)
      GOTO 5
    4 IF(FLAG.EQ.0) RETURN
      IF(CDR(K).EQ.0) RETURN
      K = CDR(K)
      L = CAR(K)
      GOTO 5
      END
```

ATOM

parameters: I1 an atom or a list

operation: ATOM checks whether I1 is a list or an atom and
returns a truthvalue indicating that.
ATOM should be declared LOGICAL in the program calling it.
NIL is considered to be a list.

code:

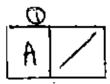
```
      LOGICAL FUNCTION ATOM (I1)
      IMPLICIT INTEGER (A-W)
      LOGICAL*1 AF
      COMMON CAR(3000),CDR(3000),AF(3000)
      ATOM = .FALSE.
      IF(AF(I1).EQ.1) ATOM = .TRUE.
      END
```

APPEND

parameters: I1, I2, I3 with I1 a pointer to a cell in a list I2 an atom or a list, I3 a pointer to another cell in a list.

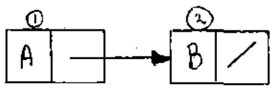
operation: APPEND creates a new cell pointed at by I3, hangs it on the CDR of I1 and puts I2 in the CAR of the new cell.

example: Given



I2 = B and I1 = 1

then after APPEND (I1, I2, I3)



with I3 = 2

code:

```
      SUBROUTINE APPEND(I1,I2,I3)
      IMPLICIT INTEGER (A-W)
      LOGICAL*1 AF
      COMMON CAR(3000),CDR(3000),AF(3000)
      COMMON /IFREE/ IFREE
      CDR(I1) = IFREE
      CAR(IFREE) = I2
      IF (IFREE.EQ.3500) GOTO 1
      I3 = IFREE
      IFREE = CDR(IFREE)
      CDR(I3) = 0
      RETURN
1     WRITE(6,2)
2     FORMAT (1X, 'STORAGE EXHAUSTED IN APPEND')
      CALL EXIT
      END
```

ATTACH

parameters: I2, I1 with I2 a list and I1 a list

operation: After the execution of ATTACH, a copy of all elements of I2 is added to I1. I2 remains available for further processing afterwards.

code:

```
          SUBROUTINE ATTACH (I2,I1)
          IMPLICIT INTEGER (A-W)
          LOGICAL*1 AF
          COMMON CAR(3000),CDR(3000),AF(3000)
          NIL = 0
          IF(I2.EQ.0) RETURN
          J = I1
C GOTO END OF LIST
    2     IF(CDR(J).EQ.NIL) GOTO 1
          J = CDR(J)
          GOTO 2
    1     K = I2
          IF(AF(I2).EQ.1) GOTO 5
C ATTACH LIST
    3     IF(K.EQ.NIL) GOTO 4
          IF(CAR(K).EQ.NIL) GOTO 6
          CALL NEW(L)
          CDR(J) = L
          J = L
          CAR(J) = CAR(K)
    6     K = CDR(K)
          GOTO 3
    4     CDR(J) = NIL
          RETURN
C ATTACH ATOM
    5     CALL NEW(K)
          CDR(J) = K
          CAR(K) = I2
          RETURN
          END
```

BACK

parameters: I, a list

operation: BACK returns one cell pointed at by I to the freelist. It is not allowed to use NIL as a parameter of BACK (this is usually the sign of a severe error in list processing). If so, the error message "NIL IN BACK" is issued and processing continues.

code:

```
          SUBROUTINE BACK(I)
          IMPLICIT INTEGER (A-Z)
          LOGICAL*1 AF
          COMMON CAR(3000),CDR(3000),AF(3000)
          COMMON /IFREE/ IFREE
C THE SUBROUTINE BACK RETURNS ONE CELL TO THE FREELIST
          IF(I.EQ.0) GOTO 10
          CDR(I) = IFREE
          CAR(I) = 0
          AF(I) = 0
          IFREE = I
          I = 0
          RETURN
   10     WRITE(6,11)
   11     FORMAT (1X, 'NIL IN BACK')
          END
```

COPY

parameters: I,a list

operation: COPY creates a new list structure equivalent to I and returns it as a value of COPY.

code:

```
      INTEGER FUNCTION COPY(I)
      IMPLICIT INTEGER (A-W)
      LOGICAL*1 AF
      COMMON CAR(3000),CDR(3000),AF(3000)
      COPY = I
      IF(I.EQ.0) RETURN
      IF(AF(I).EQ.1) RETURN
      J = I
      CALL NEW(PD5)
      CALL NEW(PD2)
      CALL NEW(COPY)
      ICO = COPY
    1 IF(AF(CAR(J)).EQ.1) GOTO 2
      IF(CAR(J).EQ.0) GOTO 2
      CALL NEW(K)
      CAR(ICO) = K
      CALL PUSH(ICO,PD2)
      CALL PUSH(J,PD5)
      ICO = CAR(ICO)
      J = CAR(J)
      GOTO 1
    2 CAR(ICO) = CAR(J)
      J = CDR(J)
      IF(J.EQ.0) GOTO 3
      CALL APPEND (ICO,0,ICO)
      GOTO 1
    3 CALL POPUP(ICO,PD2)
      CALL POPUP(J,PD5)
      IF(J.EQ.0) RETURN
      J = CDR(J)
      IF(J.EQ.0) GOTO 3
      CALL APPEND (ICO,0,ICO)
      GOTO 1
      END
```

ELEM

parameters: I1, I2 an atom and a list respectively

operation:

ELEM checks whether the atom addressed by I1 is in the list addressed by I2, if so the result is set to 1, else to 0.

```
        INTEGER FUNCTION ELEM(I1,I2)
        IMPLICIT INTEGER (A-W)
        LOGICAL*1 AF
        COMMON CAR(3000),CDR(3000),AF(3000)
        ELEM = 0
        I3 = I2
        IF(AF(I1).EQ.1) GOTO 1
        WRITE(6,2)
2       FORMAT (1X, 'FIRST ARGUMENT OF ELEM SHOULD BE ATOM')
        RETURN
1       IF(AF(I3).EQ.1) GOTO 4
5       IF(I3.EQ.0) RETURN
        IF(CAR(I3).EQ.I1) GOTO 3
        I3 = CDR(I3)
        GOTO 5
4       IF(I3.NE.I1) RETURN
3       ELEM = 1
        RETURN
        END
```

ERASE

parameters: I1 a list

operation: ERASE removes all cells used to represent a list structure and returns them to the freelist; atoms appearing in the list structure are not removed.

code:

```
            SUBROUTINE ERASE (I1)
            IMPLICIT INTEGER (A-W)
            LOGICAL*1 AF
            COMMON CAR(3000),CDR(3000),AF(3000)
            COMMON /IFREE/ IFREE
            NIL = 0
C 'ERASE' REMOVES ALL CELLS USED TO REPRESENT A LIST STRUCTURE AND RETURNS
C THEM TO THE FREELIST . MOREOVER ATOMS APPEARING IN THE LIST STRUCTURE ARE
C NOT REMOVED
            IF(AF(I1).EQ.1) RETURN
            IF(I1.EQ.0) RETURN
            CALL NEW(POS)
      3     IF(I1.EQ.0) GOTO 1
            IF (( AF(CAR(I1)).EQ.1).OR.(CAR(I1).EQ.0)) GOTO 2
            CALL PUSH(I1,POS)
            I1 = CAR(I1)
            GOTO 3
      2     I = I1
            I1 = CDR(I1)
            CDR(I) = IFREE
            CAR(I) = 0
            AF(I) = 0
            IFREE = I
            GOTO 3
      1     CALL POPUP(I1,POS)
            IF(I1.EQ.NIL) RETURN
            GOTO 2
            END
```

GET

parameters: I1, I2, I3, with I1 an atom, I2 an atom, I3 an atom or a list.

operation: GET returns the value I3 of the property I2 on the property list of the atom I1.
If I1 is not an atom, an error message is produced: "FIRST ARGUMENT OF GET SHOULD BE ATOM". If the property I2 is not on the propertylist of I3, I3 is set to NIL.

code:

```
      SUBROUTINE GET (I1,I2,I3)
      IMPLICIT INTEGER (A-Z)
      LOGICAL*1 AF
      COMMON CAR(3000),CDR(3000),AF(3000)
C CHECK WHETHER THE PROPERTY IS ALREADY THERE
      NIL = 0
      IF(AF(I1).EQ.1) GOTO 50
      WRITE(5,25)
25    FORMAT (1X, 'FIRST ARGUMENT OF GET SHOULD BE ATOM')
      CALL EXIT
50    J1 = I1
      J1 = CDR(J1)
100   IF(CDR(J1).EQ.NIL) GOTO 10
      J1 = CDR(J1)
      IF(CAR(CAR(J1)).NE.I2) GOTO 100
C  IT IS THERE
200   I3 = CDR(CAR(J1))
      RETURN
C IT IS NOT THERE
10    I3 = NIL
      RETURN
      END
```

INIT

parameters: none

operation: INIT is called at the start of any program using the FORLI library. It creates the freelist by linking the CDR cells to the next cell.

code:

```
      SUBROUTINE INIT
C THE SUBROUTINE INIT CREATES THE FREELIST
      IMPLICIT INTEGER (A-W)
      LOGICAL*1 AF
      COMMON CAR(3000),CDR(3000),AF(3000)
      COMMON /IFREE/ IFREE
C CREATE FREELIST
      DO 1 I = 1,1500
      AF(I) = 0
      CAR(I) = 0
      CDR(I) = I+1
      J = I +1500
        AF(J) =0
      CAR(J) = 0
1       CDR(J) = J+1
      DO 2 I=1,4
2     CDR(I) = 0
      IFREE = 5
      RETURN
      END
```

INPUT

parameters: IBUF, JZ, DEV

operation: INPUT is an auxiliary subroutine for the read-routines. It consumes one piece of input for the inputdevice (DEV) starting from the IBUF-th character on the input line. A new inputline is read when necessary. INPUT returns in JZ a special code if the piece of input is a punctuation mark, else JZ is the base register of an atom. INPUT constructs the necessary bookkeeping cells for atoms if the atom is a new one. INPUT calls SCAN to decode the characters and LOOKUP to consult the atom dictionary.

code:

```
          SUBROUTINE INPUT(IBUF,JZ,DEV)
          IMPLICIT INTEGER (A-W)
          LOGICAL*1 AF
          INTEGER ISTR(30),SCAN
          LOGICAL*1 STRIN
          LOGICAL*1 ALF(56)
          COMMON CAR(3000),COR(3000),AF(3000)
          COMMON /PRIN/IPRIN,BLANK,FIRST
          COMMON /STRIN/ STRIN(80)
          DATA ILEN/80/
          NIL = 0
C
C 1) CONTROL
C READ NEW BUFFER IF OLD ONE IS EXHAUSTED
    1     IF(IBUF.LT.ILEN) GOTO 2
    20    IF(DEV.NE.0) READ(DEV,3,END=20) (STRIN(I),I=1,ILEN)
          IF(DEV.EQ.0) CALL IN
    3     FORMAT (80A1)
          IF(IPRIN .EQ.1) WRITE(6,6) (STRIN(I),I=1,ILEN)
    6     FORMAT (1X, 80A1)
          IF(STRIN(1).EQ.ALF(42)) GOTO 20
          IBUF = 0
    2     IBUF = IBUF +1
          IF(STRIN(IBUF).EQ.ALF(1)) GOTO 1
```

```
C DECODE CHARACTER
          J = SCAN(IBUF)
C    SEND TO VARIOUS SUBPARTS
          IF(J.GT.3) GOTO 4
C    PUNCTUATION
          JZ = -J
          RETURN
C
C 2) ATOMS
C   (A) CHECK FOR NIL
 4        IF(J.NE.19) GOTO 12
          IF(STRIN(IBUF+1).NE.ALF(6))GOTO 12
          IF(STRIN(IBUF+2).NE.ALF(17))GOTO 12
          IJ = SCAN(IBUF+3)
          IF(IJ.GE.4) GOTO 12
          JZ = 0
          IBUF = IBUF +2
          RETURN
C   (B) ATOMS AND NUMBERS
C PREPARE FOR STORING THE CODED ATOM AND CREATE A NEW CELL (IZ) FOR CONSULTING
C THE DICTIONARY ALSO COMPUTE THE BEGINPOINT OF THE DICTIONARY
 12       K = 1
          ISTR(K) = J
          CALL NEW(IZ)
          ID = (J/10)+1
C LOOKUP BY STORING THE CODE IN IZ AND CALLING THE LOOKUP SUBROUTINE
          CDR(IZ) = 0
 8        CAR(IZ) = J
          AF(IZ) = 1
          CALL LOOKUP(ID,IZ,JZ)
          IF(JZ.EQ.0) GOTO 18
C IF NECESSARY READ NEW BUFFER FOR NEXT CHARACTER
          IF(IBUF.LT.ILEN) GOTO 9

          IF(DEV.NE.0) READ(DEV,3,END=20) (STRIN(I),I=1,ILEN)
          IF(DEV.EQ.0) CALL IN
          IF(IPRIN .EQ.1) WRITE(6,6) (STRIN(I),I=1,ILEN)
          IF(STRIN(1).EQ.ALF(42)) GOTO 20
          IBUF = 0
 9        IBUF = IBUF +1
C IF ELEMENT IN INPUT IS ATOM DELIMITER GOTO END OF ATOM ELSE GO ON WITH CON
C SULTATION OF THE DICTIONARY
          J = SCAN(IBUF)
          IF(J.LT.4) GOTO 10
          K = K +1
          IF(K.GT.30) GOTO 21
          ISTR(K) = J
          GOTO 8
C END OF ATOM
 10       IBUF = IBUF -1
          CALL BACK(IZ)
          IF(CDR(JZ).EQ.NIL) GOTO 120
C CHECK WHETHER BASE CELL OF ATOM WAS ALREADY IN DICTIONARY EITHER
C IMMEDIATELY AS CDR OF LAST CELL OR AS EMBEDDED BASE CELL
C IF NOT MAKE NEW CELL FOR EMBEDDING FOR BASE CELL AND FOR PRINTNAME
 121      J1 = JZ
          JZ = CDR(JZ)
          IF(CAR(JZ).EQ.NIL) RETURN
          IF(AF(JZ).NE.NIL) GOTO 128
          IF(CAR(CAR(JZ)).NE.NIL) GOTO 127
          JZ = CAR(JZ)
          RETURN
 127      IF(AF(CDR(JZ)).EQ.0) GOTO 121
          IF(CAR(CDR(JZ)).NE.NIL) GOTO 128
          JZ = CDR(JZ)
          RETURN
```

```
C ELSE MAKE NEW CELLS
  128  CALL NEW(I)
       J = CDR(J1)
       CDR(J1) = I
       CDR(I) = J
       CALL NEW(J)
       CAR(I) = J
       GOTO 13
  120  CALL NEW(J)
       CDR(JZ) = J
  13   AF(J) = 1
       CAR(J) = 0
       JZ = J
       CALL NEW(L)
       CDR(J) = L
       CALL NEW(J)
       CAR(L) = J
C  CODING FOR PRINTNAME
  15   L = 1
  14   AF(J) = ISTR(L)
       L = L+1
  33   IF(K-L) 30,31,32
  30   RETURN
  31   CAR(J) = ISTR(L)
       RETURN
  32   CAR(J) = (ISTR(L)*100) + ISTR(L+1)
       L = L+2
       IF((K-L+1).EQ.0) RETURN
       CALL NEW(C)
       CDR(J) = C
       J = C
       GOTO 14
C
C(3) P-ATOMS
C
C (4) ERRORS AND END OF FILE
  18   CALL BACK(JZ)
       RETURN
  20   JZ = -1
       RETURN
  21   WRITE(6,25)
  25   FORMAT (1X, 'ATOMLENGTH EXCEEDS 30 CHARACTERS')
       CALL EXIT
       END
```

LIST

parameters: I a list or an atom.

operation: LIST checks whether I is a list or an atom, and returns a truthvalue indicating that. LIST should be declared LOGICAL in the program calling it. NIL is considered to be a list.

code:

```
      LOGICAL FUNCTION LIST (II)
      LOGICAL*1 AF
      COMMON CAR(3000),CDR(3000),AF(3000)
      LIST = .FALSE.
      IF(AF(II).EQ.0) LIST=.TRUE.
      END
```

LOOKUP

parameters: ID, IZ, JY

operation: LOOKUP consults a dictionary (ID) to see whether information in a cell (IZ) is present. If so, the point in the dictionary is returned as JY, else the dictionary is extended to deal with the new information.
In addition there is a check whether the space for list cells is not exhausted. If so an error message is issued: "STORAGE EXHAUSTED DURING LOOKUP".

code:

```
          SUBROUTINE LOOKUP(ID,IZ,JY)
          IMPLICIT INTEGER (A-W)
          LOGICAL*1 AF
          COMMON CAR(3000),CDR(3000),AF(3000)
          COMMON /IFREE/ IFREE
          NIL = 0
C (1) LOOKUP
    1     IF(CDR(ID).EQ.NIL) GOTO 7
    2     IS = ID
          ID = CDR(ID)
          JY = ID
          IF(AF(ID).EQ.0) JY = CAR(ID)
    4     IF(CAR(JY).NE.CAR(IZ)) GOTO 3
          ID = JY
          RETURN
    3     IF(JY.NE.ID) GOTO 2
C (2) CREATE NEW EMBEDDING
    9     CDR(IS) = IFREE
          AF(IFREE) = 0
          CAR(IFREE) = ID
          ID = IFREE
          IF(IFREE.EQ.3000) GOTO 10
          IFREE = CDR(IFREE)
C (3) CREATE NEW CELL ON DICTIONARY
    7     CDR(ID) = IZ
          ID = IZ
          CDR(ID) = 0
          IZ = IFREE
          IF(IFREE.EQ.3000) GOTO 10
          IFREE = CDR(IFREE)
          CDR(IZ) = 0
          JY = ID
          RETURN
C (4) ERRORS
   10     WRITE(6,11)
   11     FORMAT (1X, 'STORAGE EXHAUSTED DURING LOOKUP')
          CALL EXIT
          END
```

NEW

parameters: I

operation: NEW takes one cell from the freelist and sets I equal to this cell. In addition it checks whether the memory space is exhausted and if so an error message "STORAGE EXHAUSTED IN NEW" is issued.

code:

```
          SUBROUTINE NEW(I)
          IMPLICIT INTEGER (A-W)
          LOGICAL*1 AF
          COMMON CAR(3000),CDR(3000),AF(3000)
          COMMON /IFREE/ IFREE
C THE SUBROUTINE NEW TAKES ONE CELL FROM THE FREELIST
          IF(IFREE.EQ.3000) GOTO 1
          I = IFREE
          IFREE = CDR(IFREE)
          CDR(I) = 0
          RETURN
1         WRITE(6,2)
2         FORMAT (1X, 'STORAGE EXHAUSTED IN NEW')
          CALL EXIT
          END
```

PLOTLI

parameters: I1, I, K, L

operation: PLOTLI writes a list I1 on a file on disk: FOR004.DAT in a format which can be consumed by the PLOT program. It denotes a value for the size of the characters of horizontal lines and the space between the leaves. This value is equal to I x 0.25 cm. So, if I is set to 1, the size of the characters will be 0.25 cm which is more or less the normal size. K denotes either 0 or 1. If K is 0 then the tree is not centered, if K = 1 the tree is centered, i.e. the lines from dominating nodes will end at the middle of the bar connecting the dominated nodes. L denotes either 0 or 1. If L is 0 then the leaves will 'hang' right under their dominating nodes, if L = 1 then the leaves are plotted on one line.

code:

```
      SUBROUTINE PLOTLI(I1,I,K,L)
      IMPLICIT INTEGER (A-W)
      LOGICAL*1 AF
      COMMON CAR(3000),CDR(3000),AF(3000)
      CALL PRLIST(I1,1,4)
      WRITE(4,1) I,K,L
1     FORMAT (3I2)
      RETURN
      END
```

REMARKS:

1. Files from PLOTLI are written on FOR004.DAT so do not confuse this with other output on this file by PRLIST.
2. When all structures to be plotted are processed by PLOTLI, one should call the CLOSE subroutine in the FORTRAN program, in particular CALL CLOSE (4). This is needed to 'close' the files, i.e. add an 'end of file symbol' to it.

POPUP

parameters: I, I1 with I an atom or a list and I1 a list.

operation: POPUP sets I equal to the contents of the top cell of a list I1 and then removes this cell from the top. This is done by transferring all information from the second to the first cell such that the value of I1 remains the same.

code:

```
      SUBROUTINE POPUP(I,I1)
      IMPLICIT INTEGER (A-N)
      LOGICAL*1 AF
      COMMON CAR(3000),CDR(3000),AF(3000)
      COMMON /IFREE/ IFREE
      I = CAR(I1)
      IF(CDR(I1).EQ.0) GOTO 1
      I2 = CDR(I1)
      CDR(I1) = CDR(I2)
      CAR(I1) = CAR(I2)
      AF(I1) = AF(I2)
C REMOVE SECOND CELL
      CDR(I2) = IFREE
      CAR(I2) = 0
      AF(I2) = 0
      IFREE = I2
      RETURN
    1 CALL BACK(I1)
      I1 = 0
      END
```

PROP

parameters: I1, I2, I3 with I1 an atom, I2 an atom and I3 a list or an atom.

operation: PROP appends the property I2 and the associated value I3 which may be an atom or a list to the property list of atom I1 if and only if the property is not yet on the list, else the old value is replaced by I3 without warning.
If I1 is not an atom, an error message is produced:
'FIRST ARGUMENT OF PROP SHOULD BE ATOM.'

code:

```
          SUBROUTINE PROP(I1,I2,I3)
          IMPLICIT INTEGER (A-W)
          LOGICAL*1 AF
          COMMON CAR(3000),CDR(3000),AF(3000)
    C CHECK WHETHER THE PROPERTY IS ALREADY THERE
          NIL = 0
          IF(AF(I1).NE.0) GOTO 1
          WRITE(6,5)
    5     FORMAT (1X, 'FIRST ARGUMENT OF PROP SHOULD BE ATOM')
          CALL EXIT
    1     J1 = I1
          J1 = CDR(J1)
    100   IF(CDR(J1).EQ.NIL) GOTO 10
          J1 = CDR(J1)
          IF(CAR(CAR(J1)).NE.I2) GOTO 100
    C  IT IS THERE
          CDR(CAR(J1)) = I3
          RETURN
    C IT IS NOT THERE
    10    CALL NEW(I)
          CALL APPEND (J1,I,J1)
          CAR(I) = I2
          CDR(I) = I3
          RETURN
          END
```

PRLIST

parameters: INP, BUF, DEV

operation: PRLIST prints a list or an atom.
INP is a pointer to a list (i.e. to the first element of a list) or the base register of an atom
BUF is an integer value denoting the position on the outputline from where the system should start printing; if I2 is Ø a line is left open and the system starts from the first character on the next outputline.
DEV is the device on which the output must appear, if DEV = Ø the outputline is constructed but not printed out. This is of use in extracting the printname of atoms via commonzones.
The result of PRLIST is that the whole list structure pointed at by INP is recoded in alphanumeric characters and transferred to the device.

remarks: 1.If list notation is impossible, dot notation is used but only at the point where it is necessary:
E.g. given (A . (B . (C . D))) , this will be printed as (A B C . D).
2. When the value of BUF is greater than one, all characters on the outputline are blanks. One can use this feature for editing. E.g. suppose you want the following as output:
THE NAME IS : JOHN, where "the name is:"is in the program and John an atom referred to by the variable name, then the output can be obtained by the following lines of FORTRAN:

```
      CALL PRLIST (NAME, 14, 6)
      WRITE (6,1)
    1 FORMAT (1H+, 'THE NAME IS :')
```

code

code:

```
      SUBROUTINE PRLIST(INP,BUF,DEV)
      IMPLICIT INTEGER (A-W)
      LOGICAL*1 AF
      LOGICAL*1 STRIN
      LOGICAL*1 ALF(56)
      COMMON /STRIN/STRIN(80)
      COMMON CAR(3000),CDR(3000),AF(3000)
      DATA ILEN/70/
C THIS SUBROUTINE PRINTS A LIST POINTED AT BY INP ON A DEVICE CALLED DEV
C FROM THE POSITION INDICATED BY BUF
      NIL = 0
      IBUF = BUF
      IF(BUF.LE.1) GOTO 400
      DO 401 I = 1,BUF
401   STRIN(I) = ALF(1)
400   IF((DEV.EQ.0).OR.(IBUF.NE.0))GOTO 402
      WRITE(DEV,403)
403   FORMAT (1X/1X)
402   I1 = INP
      IF (IBUF.EQ.0) IBUF = 1
      IOUT = 1
C TOP CONTROL : SEE WHETHER INPUT IS ATOM,NIL, OR LIST
      IF(AF(I1).EQ.1) GOTO 100
      IF(I1.EQ.0) GOTO 200
C IF LIST CREATE PEG CELL ON TOP OF LIST
      IOUT = 0
      I = I1
      CALL NEW(I1)
      CAR(I1) = I
      CALL NEW(POS)
      GOTO 2
C
C NORMAL CONTROL
C ---------------
3     I1 = CDR(I1)
2     IF(I1.EQ.NIL) GOTO 114
      IF(AF(I1).EQ.0) GOTO 1
      IF(CAR(I1).NE.0) GOTO 8
C
C SECTION 1 PRINTING THE ATOMS
C-------------------------------
C GOTO PRINTNAME CELL OF ATOM, DECODE THE PRINTNAME AND WRITE IT ON THE OUT-
C PUTBUFFER    (STRIN)
100   I1 = CDR(I1)
      PRN = I1
15    I1 = CAR(PRN)
      IBEG = IBUF -1
11    IF(IBUF+1.LT.ILEN) GOTO 14
      IF(DEV.NE.0) WRITE(DEV,6) (STRIN(I),I=1,IBEG)
6     FORMAT (1X, 120A1)
      IBUF = 1
      GOTO 15
14    STRIN(IBUF) = ALF(AF(I1))
      I2 = CAR(I1)
      IF(I2.EQ.0) GOTO 12
      IBUF = IBUF +1
      IF(I2.LT.100) GOTO 16
```

```
        STRIN(IBUF) = ALF(I2/100)
        IBUF = IBUF +1
        I2 = I2 - ((I2/100) * 100)
 16     STRIN(IBUF) = ALF(I2)
        IF(CDR(I1).EQ.0) GOTO 12
        I1 = CDR(I1)
        IBUF = IBUF +1
        GOTO 14
C END OF ATOM OR P-ATOM / ADD BLANK
 12     IBUF = IBUF +1
        STRIN(IBUF) = ALF(1)
        IF(IBUF.GT.ILEN) GOTO 11
        IBUF = IBUF +1
        IF(IOUT.EQ.1) GOTO 110
        GOTO 114
C
C LEFT PARENTHESIS
C ------------------
C PUSH POINTER TO CURRENT CELL AND SET CURRENT CELL EQUAL TO CAR , ADD LEFT
C PARENTHESIS IF EMBEDDING NOT DUE TO AN ATOM
 1      IF(CAR(I1).EQ.NIL) GOTO 200
        CALL PUSH(I1,PDS)
        IF(AF(CAR(I1)).EQ.1) GOTO 117
        IF(IBUF.LT.ILEN) GOTO 19
        IBUF = IBUF -1
        IF(DEV.NE.0) WRITE(DEV,6) (STRIN(I),I=1,IBUF)
        IBUF = 1
 19     STRIN(IBUF) = ALF(2)
        IBUF = IBUF +1
 117    I1 = CAR(I1)
        GOTO 2
C
C RIGHT PARENTHESIS
C------------------
C POPUP POINTER TO CURRENT CELL AND ADD RIGHT PARENTHESIS IF EMBEDDING IS
C NOT DUE TO AN ATOM , IF THE PUSHDOWN STORE IS EMPTY GOTO END
 114      CALL POPUP(I1,PDS)
        IF(CAR(PDS).EQ.NIL) GOTO 300
        IF((AF(CAR(I1)).EQ.1).OR.(CAR(I1).EQ.NIL)) GOTO 16
        IF(IBUF.LT.ILEN ) GOTO 22
        IBUF = IBUF -1
        IF(DEV.NE.0) WRITE(DEV,6) (STRIN(I),I=1,IBUF)
        IBUF = 1
 22     STRIN(IBUF) = ALF(3)
        IBUF = IBUF +1
 18     IF(CDR(I1).EQ.NIL) GOTO 114
C DOT
C IF IN THE CDR THERE IS A POINTER TO AN ATOM WE ADD A DOT
        IF(AF(CDR(I1)).EQ.NIL) GOTO 3
        IF (IBUF.LT.ILEN) GOTO 23
        IBUF = IBUF -1
        IF(DEV.NE.0) WRITE(DEV,6) (STRIN(I),I=1,IBUF)
        IBUF = 1
 23     STRIN(IBUF) = ALF(55)
        STRIN(IBUF+1) = ALF(1)
        IBUF = IBUF +2
        GOTO 3
```

```
C NIL
200    IF(IBUF+2.LT.ILEN) GOTO 210
       IBUF = IBUF -1
       IF(DEV.NE.0) WRITE(DEV,6) (STRIN(I),I=1,IBUF)
       IBUF = 1
210    STRIN(IBUF) = ALF(19)
       STRIN(IBUF+1) = ALF(6)
       STRIN(IBUF+2) = ALF(17)
       STRIN(IBUF+3) = ALF(1)
       IBUF = IBUF +4
       IF(IOUT.EQ.1) GOTO 110
       GOTO 3
C END
300    STRIN(IBUF) = ALF(3)
       IBUF = IBUF +1
110    IF(FLAG.EQ.1) RETURN
500    IBUF = IBUF -1
       IF(DEV.NE.0) WRITE(DEV,6) (STRIN(I),I=1,IBUF)
       IBUF = IBUF +1
       RETURN
C ERROR
8       WRITE(6,88)
88     FORMAT (1X, 'IRREGULAR INPUT FOR PRLIST(POSSIBLY PART OF DICTIONARY
      *)')
       RETURN
       END
```

PUSH

parameters: I, I1, with I a list or an atom and I1 a list.

operation: PUSH creates a new cell on top of a list pointed at by I1 and sets I in the CAR of this cell. the value of the pointer itself does not change during PUSH, because actually the second cell becomes the new cell and all information on the former first cell is transferred to this cell.

code:

```
      SUBROUTINE PUSH(I,I1)
      IMPLICIT INTEGER (A-W)
      LOGICAL*1 AF
      COMMON CAR(3000),CDR(3000),AF(3000)
      COMMON /IFREE/ IFREE
C THIS SUBROUTINE CREATES A NEW CELL ON TOP OF A LIST I1 AND STORES I IN
C THE CAR OF THIS CELL
      IF(I1.EQ.0) GOTO 3
7     I2 = I1
C TRANSFER INFORMATION OF FIRST CELL TO NEW CELL
      IF(IFREE.EQ.3000) GOTO 1
      I1 = IFREE
      IFREE = CDR(IFREE)
      AF(I1) = AF(I2)
      CAR(I1) = CAR(I2)
      CDR(I1) = CDR(I2)
C STORE NEW INFORMATION IN TOP CELL
      AF(I2) = 0
      CAR(I2) = I
      CDR(I2) = I1
      I1 = I2
      RETURN
1     WRITE(6,2)
2     FORMAT (1X, 'STORAGE EXHAUSTED IN PUSH')
      CALL EXIT
3     CALL NEW(I1)
      GOTO 7
      END
```

RLIST

parameters: BUF, IBUF, DEV

operation: RLIST is an integer function for reading lists and atoms.
BUF is a pointer to the position where the reading should start.
IBUF is a pointer which results in the final position after executing the function.
DEV is a code for the device from which the system should read.
The result of RLIST is that all decoding and storing is performed and that a pointer to a list (or atom) is returned as result.

The following conventions hold for the arguments:
1. If BUF is equal to Ø, then a new line of input is consumed but the line is NOT printed out during reading.
If BUF is equal to 1, a new line of input is consumed and the line is printed on the output device (LUN: 6).
If BUF is greater than 1, the system starts reading on the latest consumed line.
Whenever a line is completely processed, but more characters are needed, the system keeps reading new lines from the input device until a complete list (or atom) is found.

2. IBUF is set to the final character used in the RLIST process. So, with IBUF we can keep on reading on the same line if we take this as starting point for the next call to RLIST.

3. DEV indicates the device from which the input line must be taken.
if DEV =Ø a special subroutine called IN is used to fill the characters of the intputline in the commonzone STRIN. The user can himself define the way in which this filling in is performed.
If DEV is greater than Ø the relevant device should during taskbuilding be connected to the logical unit number specified in DEV.

Remarks: 1. Blanks are ignored if not meaningful

2. Superfluous right brackets on the last inputline are ignored but if you keep reading on the same line, an error message will follow: 'TOO MANY RIGHT PARENTHESES'.

3. A lack of right brackets will make the system look for further brackets and therefore consume the rest of input lines. Then a message will be issued: 'TOO MANY LEFT PARENTHESES'. So, a lack of right brackets is a fatal error, in that it is noticed only when all cards have been read.

4. The null string can be representedin the input by NIL and (). NIL is the only atom that is present as soon as the program starts. (The integer value of NIL is Ø).

5. Each character that is given as input is coded directly into an integer. Characters which are not in the ALF vector are not accepted, a message 'UNRECOGNIZED CHARACTER' is issued.

6. An important (but difficult) question is the fact that there is a fundamental distinction between the FORTRAN program and the variables for lists and atoms used therein and the users' specification for the atoms and lists, a distinction which is not so stringent in LISP e.g., due to the QUOTE-feature. Clearly the bridge between the two is the RLIST function. Therefore any atom that is used as an entity in the program should be read in by RLIST.
E.G. suppose 'NOUN' is an entity which is being referred to in the program, then we can write

NOUN = RLIST (1,I,1) where NOUN is on the card.

From then on the variable 'NOUN' (in the FORTRAN program) will refer to the same object as the atom NOUN in input/output.

code:

```
      INTEGER FUNCTION RLIST (BUF,IBUF,DEV)
C
C (1) START
C --------
      IMPLICIT INTEGER (A-W)
      LOGICAL*1 AF
      LOGICAL*1 ALF(56),STRIN
      COMMON /STRIN/ STRIN(80)
      COMMON CAR(3000),CDR(3000),AF(3000)
      COMMON /PRIN/IPRIN,BLANK,FIRST
      DATA ALF/ ' ','(',')','A','E','I','O','U','B','C','D','F','G'
     *,'H' ,'J','K','L','M','N','P','Q','R','S','T','V','W','X','Y','Z',
     *'0','1','2','3','4','5','6','7','8','9','"',',','$','/','-',
     *'+','^',']','>','<',':',';','?','"','=','.','*'/
C FOR CONTROLING THE INPUT A BUFFERPOINTER (IBUF) IS USED WHICH POINTS TO THE
C FIRST CHARACTER TO BE READ. IBUF INITIALLY ALSO REGULATES THE PRINTFLAG
C (IPRIN) WHICH IS SET TO 1 IF THE INPUTLINE IS TO BE PRINTED OUT,ELSE TO 0
      NIL = 0
      IBUF = BUF
      RLIST = 0
      IF(IBUF.GT.1) GOTO 100
      IF(IBUF.EQ.0) IPRIN=0
      IF(IBUF.EQ.1) IPRIN=1
      IBUF = 81
C DECODE THE FIRST INPUT ELEMENT . IF IT IS A LEFT OR RIGHT PARENTHESIS
C WE START PROCESSING FURTHER, ELSE AN ATOM IS DISCOVERED AND WE IMME
C DIAZTELY RETURN WITH RLIST AS POINTER TO THE BASE CELL OF THE ATOM
100   CALL INPUT(IBUF,JZ,DEV)
      IF(JZ.EQ.-1) GOTO 24
      IF(JZ.LT.0) GOTO 1
      RLIST=JZ
      RETURN
C WHEN THE FIRST ELEMENT IS A LEFT PARENTHESIS (CODE = -3) AN ERROR OCCURRED
C ELSE WE CREATE A NEW TOPCELL AND GOTO THE CONTROL POINT
1     IF(JZ.EQ.-3) GOTO 22
      CALL NEW(RLIST)
      CALL NEW(IL)
      IR = RLIST
      GOTO 11
C
C (2) MAIN PROGRAM
C     ------------
C A NEW ELEMENT IS TAKEN FROM THE INPUT
7     CALL INPUT(IBUF,JZ,DEV)
C CONTROL POINT
C----------------
C SEND TO SECTION FOR ATOMS OR LEFT OR RIGHTPAR DEPENDING ON THE RESULT
C OF 'INPUT' . IF INPUT RESULTS IN -1 ( = END OF FILE) AN ERROR OCCURRED
      IF(JZ.GT.0) GOTO 10
11    J = JZ+4
      GOTO (9,3,20),J
C
C SECTION 1 ATOMS
C
C WHEN THE ATOM IS NIL , FIRST STORE -1
2     JZ = -1
```

```
C IF THE CAR OF THE CURRENT CEL (IR) IS EMPTY WE CAN IMMEDIATELY STORE THE AT
C ELSE A NEW CELL MUST BE MADE , AND THEN THE ATOM IS STORED
C (NOTE THE PROVISION FOR NIL)
 10     IF(CAR(IR).EQ.NIL) GOTO 5
        IF(CAR(IR).EQ.-1) CAR(IR ) = 0
        CALL NEW(I)
        CDR(IR) = I
        IR = I
 5      CAR(IR) = JZ
        GOTO 7
C
C SECTION 2 LEFT PARENTHESIS
C WHEN THE CAR OF THE CURRENT CEL IS NOT EMPTY WE FIRST CREATE A NEW CELL AND
C HANG IT ON THE ALREADY OBTAINED LIST
 3      IF(CAR(IR).EQ.NIL) GOTO 6
        IF(CAR(IR).EQ.-1) CAR(IR) = 0
        CALL NEW(I)
        CDR(IR) = I
        IR = I
C THEN/ELSE WE PUSH THE CURRENT CELL ON IL (THE PUSHDOWNSTORE) , CREATE A NEW
C CELL AND HANG IT IN THE CAR OF THE CURRENT CELL , THIS LAST CELL IS
C THE NEW CURRENT CEL
 6      CALL PUSH(IR,IL)
        CALL NEW(I)
        CAR(IR) = I
        IR = I
        GOTO 7
C
C SECTION 3 RIGHT PARENTHESIS
C CLOSE THE LIST DOWN ( = NIL IN CDR OF CURRENT CELL) AND POPUP FROM IL
C THE POINTER TO WHERE THE INBEDDING STARTED, NOTE THE PROVISION ,
C FOR NIL
 4      CDR(IR) = 0
        IF(CAR(IR).EQ.NIL) GOTO 9
        IF(CAR(IR).EQ.-1) CAR(IR) = 0
        CALL POPUP(IR,IL)
        IF(CAR(IL).NE.NIL) GOTO 7
C  END
C IF THE PUSHDOWN IS EMPTY WE REACHED THE END OF A LIST AND GO BACK TO
C THE CALLING PROGRAM
        K = RLIST
        RLIST = CAR(RLIST)
        CALL BACK(K)
        CALL BACK(IL)
        RETURN
C IN THE CASE OF NIL AS () THE CELL DUE TO EMBEDDING IS RETURNED TO THE
C FREELIST AND -1 IS STORED IN THE CAR OF THE NEW CURRENT CELL OBTAINED BY
C POPPING UP FROM THE PUSHDOWN
 9      CALL BACK(IR)
        CALL POPUP(IR,IL)
        CAR(IR) = -1
        IF(IR.NE.RLIST) GOTO 7
        CALL BACK(IL)
        CALL BACK(RLIST)
        RETURN
C
C(3) ERRORS
C ---------
 20     WRITE(6,21)
 21     FORMAT (1X,'MISSING RIGHT PARENTHESIS')
        CALL EXIT
 22     WRITE(6,23)
 23     FORMAT (1X, 'MISSING LEFT PARENTHESES')
        CALL EXIT
 24     WRITE(6,25)
 25     FORMAT (1X, 'END OF FILE DURING INPUT')
        RLIST = 0
        RETURN
        END
```

The library of list processing routines contains also a number of routines necessary to plot tree structures on the plotter. These routines, although very interesting in themselves, will not be discussed here, partly because it is a superfluous feature, partly because they make extensive use of the special UIA library containing routines for using the plotter.

3.2. THE IMPLEMENTATION OF THE PARSER

We now start with an explicit documentation of the implementation of the parser. As every programmer knows it is always possible to make other implementations for the same problem or to construct programs in other programming languages. One of the things we want to do in the near future is to implement the parser in another programming language. This is to say that we do not insist on the present implementation nor on the programming language being used, although it must be said that the system works now very efficiently and very fast.

The presentation contains three parts. First we discuss some auxiliary (but task oriented) routines such as the consultation of the dictionary, the implementation of the feature complex calculus and the implementation of the completion automata. These routines have a general character because they are called at several places during the program.

In a second part we discuss the programs which constitute the parsing system itself. In a final part we provide all details on the routines for computing functional structures, case structures and semantic structures.

3.2.1. Auxiliary routines

3.2.1.1. Storing and retrieving linguistic information

Because we are experimenting with a rather small computer, we need to store the lexicon and other kinds of linguistic information on an external storage device (a disk) although this slows the whole process down considerably.

We will solve this (largely mere technical) problem as follows. We assume that linguistic information is always related to a particular atom. E.g. in the lexicon the information sequence is associated with a particular word form, a syntactic network is associated with a particular keyword, a case frame is associated with a predicate, etc.

As a consequence we organize the file on disk in such a way that via an atom we can retrieve the information relevant for that atom. Note however that we assume there to be only one sequence of information for one atom .

The list of atoms is stored and retrieved on the basis of a hashcode which guarantees fast lookup. Because we want more then one language as 'working language', the language is a factor in the retrieval.

The routines for creating dictionaries and for retrieving information from them will now be discussed in some detail. The implementation is largely due to L. Bamps.

INI

operation:

This main program initializes two files on disk. One for the information in the dictionary (INFO.DAT) and one for the words themselves (WORD.DAT). Then the files are filled with blanks. Space is provided for 5000 information items.

code:

```
          LOGICAL*1 BL
          DATA BL/' '/
          CALL ASSIGN(4,'INFO.DAT',0)
          CALL FDBSET(4,'UNKNOWN')
          DEFINE FILE 4 (5001,41,U,IREC)
          CALL ASSIGN(3,'WORD.DAT',0)
          CALL FDBSET(3,'UNKNOWN')
          DEFINE FILE 3(7993,17,U,IREC)
          IO=0
          DO 100 I=1,7993
      100 WRITE(3'I)(BL,J=1,31),IO
          DO 101 I=1,5001
  101     WRITE(4'I)IO,(BL,J=1,80)
          CALL EXIT
          END
```

CRE

operation:

This main program creates a dictionary by reading the atoms and storing the information about the atoms.

code:

```
      LOGICAL*1 WORD(3Ø),TA,WORDH(3Ø),TAH,HW(2),BL
      LOGICAL*1 KAART(8Ø)
      EQUIVALENCE (IA,HW(1))
      DATA BL/' '/
      DATA NUL/Ø/
      CALL ASSIGN(3,'WORD.DAT',Ø)
      CALL FDBSET(3,'UNKNOWN')
      DEFINE FILE 3(7993,17,U,IREC)
      CALL ASSIGN(4,'INFO.DAT',Ø)
      CALL FDBSET(4,'UNKNOWN')
      DEFINE FILE 4(5ØØ1,41,U,IREC)
  29Ø READ(1,99,END=3ØØ)WORD,TA
   99 FORMAT(8ØA1)
      WRITE(6,98)TA,WORD
   98 FORMAT('Ø',5X,A1,5X,3ØA1)
      HW(1)=WORD(2)
      HW(2)=WORD(3)
      IAD=MOD(IA,7993)
  2ØØ IAD=IAD+1
      IF(IAD.GT.7993)IAD=1
      READ(3'IAD)WORDH,TAH,INDH
      IF(WORDH(1).EQ.BL)GO TO 25Ø
      DO 1ØØ I=1,3Ø
      IF(WORDH(I).NE.WORD(I))GO TO 2ØØ
  1ØØ CONTINUE
      IF(TA.NE.TAH)GO TO 2ØØ
      WRITE(6,97)
   97 FORMAT('++')
      INDX=INDH
  2Ø5 READ(1,99)KAART
      WRITE(6,95) (KAART(I),I=1,8Ø)
 95   FORMAT (2ØX,8ØA1)
      READ(4'INDX)INDH
      IF(KAART(8Ø).EQ.BL)GO TO 22Ø
      INDXH=-IAD
      WRITE(4'INDX)INDXH,(KAART(I),I=1,8Ø)
  21Ø IF(INDH.LT.Ø)GO TO 29Ø
      READ(4'INDH)INDXH
      WRITE(4'INDH)NUL
      INDH=INDXH
      GO TO 21Ø
  22Ø IF(INDH.GT.Ø)GO TO 23Ø
```

```
            READ(4'5001)INDL
            INDL=INDL+1
            WRITE(4'INDX)INDL,(KAART(I),I=1,80)
            INDX=INDL-1
            GO TO 251
      230   WRITE(4'INDX)INDH,(KAART(I),I=1,80)
            INDX=INDH
            GO TO 265
       250 READ(4'5001)INDX
            INDXH=INDX+1
            WRITE(3'IAD)WORD,TA,INDXH
       251 INDX=INDX+1
            READ(1,99)KAART
            WRITE(6,95)  (KAART(I),I=1,80)
            INDXH=-IAD

            IF(KAART(80).EQ.BL)INDXH=INDX+1
            WRITE(4'INDX)INDXH,(KAART(I),I=1,80)
            IF(KAART(80).EQ.BL) GOTO 251
            WRITE(4'5001)INDX
            GO TO 290
      300   CONTINUE
            END
```

SEARCH

parameters: I1 an atom

operation:

The integer function SEARCH consults the dictionary on the external storage device to find the information associated with a particular atom (I1) for a particular language (TA). If no information is in the dictionary an error message will be issued: 'LINGUISTIC INFORMATION MISSING FOR :' . This is a fatal error.

code:

```
          INTEGER FUNCTION SEARCH (I1)
          IMPLICIT INTEGER (A-W)
          LOGICAL*1 STRIN(80),WORD(30),HW(2),WORDH(30),TAH,TA,BL
          EQUIVALENCE (IA,HW(1))
          COMMON /IND/ INDX
          COMMON /TA/ TA
          COMMON /STRIN/STRIN
          DATA BL/1H /
          CALL GET(I1,-1,SEARCH)
          IF(SEARCH.NE.0) RETURN
          CALL PRLIST(I1,1,0)
          DO 1 LEN=1,30
          IF(STRIN(LEN).EQ.BL) GOTO 2
  1       WORD(LEN) = STRIN(LEN)
  2       DO 3 J = LEN,30
  3       WORD(J) = BL
          HW(1)=WORD(2)
          HW(2)=WORD(3)
          IAD=MOD(IA,7993)
      400 IAD=IAD+1
          IF(IAD.GT.7993)IAD=1
          READ(3'IAD)WORDH,TAH,INDH
          IF(WORDH(1).NE.BL)GO TO 401
          CALL PRLIST(I1,37,6)
          WRITE(6,4)
  4       FORMAT (1H+,'LINGUISTIC INFORMATION MISSING FOR :')
          CALL EXIT
      401 DO 402 I=1,30
          IF(WORDH(I).NE.WORD(I))GO TO 400
      402 CONTINUE
          IF(TA.NE.TAH)GO TO 400
          INDX=INDH
          SEARCH = RLIST(0,I,0)
           CALL PROP(I1,-1,SEARCH)
          RETURN
          END
```

IN

parameters: none

operation:

This subroutine fills the STRIN-vector in the commonzone for consumation by RLIST by reading items from disk. This is an auxiliary subroutine for the SEARCH operation.

```
      SUBROUTINE IN
      LOGICAL*1 STRIN(80),BL,TEXT(80)
      COMMON /IND/INDX
      COMMON /STRIN/ STRIN
      DATA BL/1H /
      READ(4'INDX) INDX,TEXT
      DO 1 I = 1,80
    1 STRIN(I) = TEXT(I)
      RETURN
      END
```

3.2.1.2. The implementation of the feature complex calculus

To implement the comparing and combination of feature complexes as defined in chapter I, we need routines for computing set interpretations, doing truthlogical interpretations and combinations of features. For this purpose we introduce the following programs:

EXT

parameters: GOAL (a feature complex)

operation:

The integer function EXT takes a feature complex GOAL and returns the set interpretation as value of EXT.

explanations:

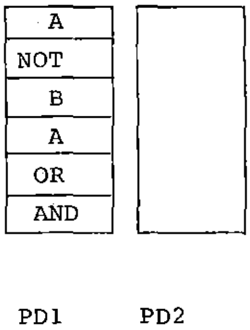
Due to the recursive nature of the set-interpretation, we will need pushdownstores to stimulate the recursivity not present in FORTRAN.
The first phase of the program consists in decomposing the whole feature complex into minimal units, where a minimal unit is an atom or an operator. Two pushdown stores are used for this PD1 to push the minimal units upon and PD2 to run through the list structure of the feature complex. E.g.
after phase 1 the feature complex (AND (OR A B) (NOT A)) becomes:

PD1 :

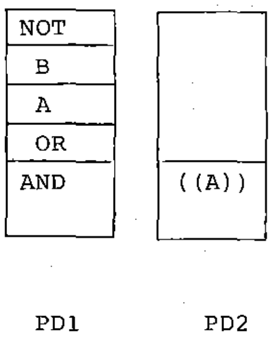
A
NOT
B
A
OR
AND

The second phase of the program takes each of these minimal units from PD1 and evaluates them. The result of evaluation is stored on PD2 and if results of previous evaluation is needed, it is taken from this pushdownstore PD2.
E.G.:

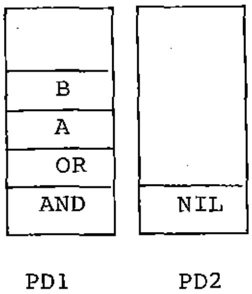
(1)


A
NOT
B
A
OR
AND
PD1
PD2

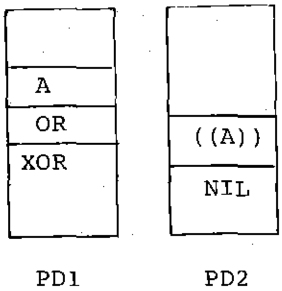
(2)


NOT
B
A
OR
AND
((A))
PD1
PD2

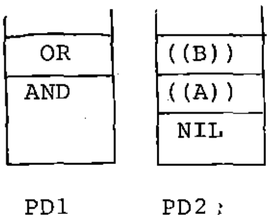
(3)


B
A
OR
AND
NIL
PD1
PD2

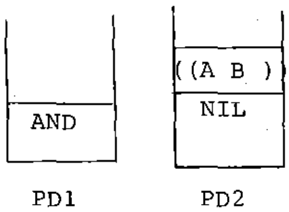
(4)


A
OR
XOR
((A))
NIL
PD1
PD2

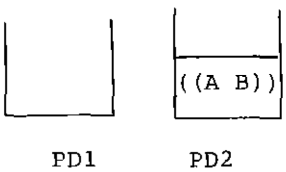
(5)


OR
AND
((B))
((A))
NIL
PD1
PD2

(6)


AND
((A B))
NIL
PD1
PD2

(7)


((A B))
PD1
PD2

end

code:

```
        INTEGER FUNCTION EXT(GOAL)
        IMPLICIT INTEGER (A-X)
        LOGICAL*1 AF
        COMMON CAR(3000),CDR(3000),AF(3000)
        COMMON /LOG/ AND,OR,XOR,NOT
        EXT = 0
        I = GOAL
        IF(GOAL.EQ.0) RETURN
C FEATURE COMPLEX IS ATOM
        IF(AF(I).NE.1) GOTO 110
        CALL NEW(EXT)
        CALL NEW(K)
        CAR(EXT) = K
        CAR(K) = I
        RETURN
C FEATURE COMPLEX IS LIST
C PHASE 1
C DECOMPOSE FEATURE COMPLEX AND PUSH ON PD1
110     CALL NEW(PD1)
        CALL NEW(PD2)
2       IF((AF(CAR(I)).EQ.1).OR.(CAR(I).EQ.0)) GOTO 1
        CALL PUSH(I,PD2)
        I = CAR(I)
        GOTO 2
1       CALL PUSH(CAR(I),PD1)
22      I = CDR(I)
        IF(I.EQ.0) GOTO 3
        GOTO 2
3       IF(CAR(PD2).EQ.0) GOTO 5
        CALL POPUP(I,PD2)
        GOTO 22
4       CALL PUSH(I,PD1)
C PHASE 2
5       IF(CDR(PD1).EQ.0) GOTO 30
        CALL POPUP(J,PD1)
C SEND TO RELEVANT PART
        IF(J.EQ.0) GOTO 9
        IF(J.EQ.NOT) GOTO 19
        IF(J.EQ.OR) GOTO 11
        IF(J.EQ.AND) GOTO 11
        IF(J.EQ.XOR) GOTO 13
C ATOM
        CALL NEW(I)
        CALL NEW(L)
        CAR(I) = L
        CAR(L) = J
        CALL PUSH(I,PD2)
        GOTO 5
C NIL
9       CALL PUSH(0,PD2)
        GOTO 5
C NOT
19      CAR(PD2) = [illegible]
        GOTO 5
C OR / AND
11      CALL NEW(LI)
        K = LI
```

```
      CALL POPUP(J,PD2)
      IF(J.EQ.0) GOTO 5
      I = CAR(PD2)
      IF(I.EQ.0) GOTO 113
      J1 = J
10    EJ = CAR(J1)
      I1 = I
12    EI = CAR(I1)
      S = COPY(EJ)
      CALL APPEND (K,S,K)
      CALL ADD(EI,S)
      I1 = CDR(I1)
      IF(I1.NE.0) GOTO 12
      J1 = CDR(J1)
      IF(J1.NE.0) GOTO 10
      CAR(PD2) = CDR(LI)
      CALL BACK(LI)
      GOTO 5
123   CAR(PD2) = L
      GOTO 5
C XOR
113   CAR(PD2) = J
      GOTO 5
13    CALL POPUP(L,PD2)
      IF(L.EQ.0) GOTO 5
      I = CAR(PD2)
      IF(I.EQ.0) GOTO 123
210   CALL PUSH(CAR(I),L)
      K = CDR(I)
      CALL BACK(I)
      I = K
      IF(I.NE.0) GOTO 210
      CAR(PD2) = L
      GOTO 5
C END
30    CALL POPUP(EXT,PD2)
      CALL BACK(PD1)
      CALL BACK(PD2)
      RETURN
      END
```

MATCH

parameters: SOURCE, GOAL two feature complexes where GOAL is a set-interpretation;
INFTR an inference tree

operation: The integer function MATCH computes the subsets of the domain (given by GOAL) which evaluate to true for the feature complex source and returns the set of these subsets as the value of match.

explanations:
MATCH works on the same principles as EXT except as regards the evaluation procedure itself.
In a first phase the feature complex is decomposed in minimal units and stored on the pushdownstore PD1. The other pushdownstore PD2 is used to assist in scanning through the structure.
The second part is the evaluation itself. Here we make use of a special subroutine MATCH2 that checks whether an atom is in a subset which is itself a part of the feature complex GOAL.
The whole process is repeated for as many subsets as there are in the domain, and the subsets which result in true are accumulated and returned as final result.
The code for the truthvalues is 1 for true and -1 for false.

code:

```
            INTEGER FUNCTION MATCH (SOURCE,GOAL)
            IMPLICIT INTEGER (A-X)
            LOGICAL*1 AF
            COMMON CAR(3ØØØ),CDR(3ØØØ),AF(3ØØØ)
            COMMON /LOG/AND,OR,XOR,NOT
            CALL NEW(IM)
            M = IM
            K = GOAL
     2Ø     IF(K.EQ.Ø) IK = Ø
            IF(K.NE.Ø) IK = CAR(K)
      C PHASE (1)
      C DECOMPOSE FEATURE COMPLEX AND PUSH ON PD1
            I = SOURCE
            CALL NEW(PD1)
            CALL NEW(PD2)
            IF(I.EQ.Ø) GOTO 4
            IF(AF(I).EQ.1) GOTO 4
     2      IF((AF(CAR(I)).EQ.1).OR.(I.EQ.Ø)) GOTO 1
            CALL PUSH(I,PD2)
            I = CAR(I)
            GOTO 2
```

```
 1      CALL PUSH(CAR(I),PD1)
 22     I = CDR(I)
        IF(I.NE.0) GOTO 2
 3      IF(CAR(PD2).EQ.0) GOTO 5
        CALL POPUP(I,PD2)
        GOTO 22
 4      CALL PUSH(I,PD1)
 C PHASE(2)
 5      IF(CDR(PD1).EQ.0) GOTO 30
        CALL POPUP(J,PD1)
 C SEND TO RELEVANT PARTS
        IF(J.EQ.0) GOTO 9
        IF(J.EQ.NOT) GOTO 10
        IF(J.EQ.OR) GOTO 11
        IF(J.EQ.AND) GOTO 12
        IF(J.EQ.XOR) GOTO 13
 C ATOMS
        CALL PUSH(MATCH2 (J,IK),PD2)
        GOTO 5
 C NIL
 9      CALL PUSH(1,PD2)
        GOTO 5
 C NOT
 10     IF(CAR(PD2).EQ.0) GOTO 40
        CAR(PD2) = CAR(PD2)*-1
        GOTO 5
 C OR
 11     IF(CAR(PD2).EQ.0) GOTO 40
        CALL POPUP(L,PD2)
        IF(L.EQ.1) CAR(PD2) = 1
        GOTO 5
 C AND
 12       IF(CAR(PD2).EQ.0) GOTO 40
        CALL POPUP(L,PD2)
        IF(L.EQ.-1) CAR(PD2) = -1
        GOTO 5

 C XOR
 13     IF(CAR(PD2).EQ.0) GOTO 40
        CALL POPUP(L,PD2)
        IF(L.EQ.1) GOTO 33
        IF(CAR(PD2).EQ.1) GOTO 5
 34     CAR(PD2) = -1
        GOTO 5
 33     IF(CAR(PD2).EQ.1) GOTO 34
        CAR(PD2) = 1
        GOTO 5
 40     WRITE(6,41)
 41     FORMAT (1X, 'UNWELLFORMED FEATURE COMBINATION IN MATCH TEST')
        MATCH = -1
        GOTO 31
 30     IF(CAR(PD2).EQ.0) GOTO 40
        CALL POPUP(MATCH,PD2)
        IF(CAR(PD2).NE.0) GOTO 40
 C ACCUMULATE RESULTS AND END
 31     CALL BACK(PD2)
        CALL BACK(PD1)
        IF(MATCH.EQ.1) CALL APPEND (IM,IK,IM)
        K = CDR(K)
        IF(K.NE.0) GOTO 20
 25     MATCH = 0
        IF(CDR(M).EQ.0) RETURN
        MATCH = CDR(M)
        CALL BACK(M)
        RETURN
        END
```

MATCH2

parameters: J; IK with J an atom and IK a linear list
INFTR an inference tree.

operation:
The integer function MATCH2 checks whether the atom J is in the list IK. If so, MATCH2 is set to 1, else to -1.

code:

```
        INTEGER FUNCTION MATCH2(J,IK,INFT)
        IMPLICIT INTEGER (A-Z)
        LOGICAL*1 AF
        COMMON CAR(3000),CDR(3000),AF(3000)
        MATCH2=-1
        K = IK
1       IF(K.EQ.0) GOTO 11
        IF(CAR(K).EQ.J) GOTO 10
        IF(INFT.EQ.0) GOTO 2
        IF(CROSS(J,CAR(K),INFT).NE.0) GOTO 10
2       K = CDR(K)
        GOTO 1
10      MATCH2 = 1
        RETURN
11      IF(J.NE.0) RETURN
        GOTO 10
        END
```

CROSS

parameters: SOU and GOAL both atoms,
INFTR an inference tree.

operation:

The integer function CROSS is an auxiliary subroutine for MATCH2, it computes whether two atoms can be related to each other on the basis of an inference tree. This is done by running through the inference tree (with a pointer LI) using a pushdownstore (PDS) and by setting flags at relevant points during scanning.

code:

```
         INTEGER FUNCTION CROSS (SOU,GOAL,INFTR)
          IMPLICIT INTEGER(A-W)
          LOGICAL*1 AF
          COMMON CAR(3000),CDR(3000),AF(3000)
          CROSS =0
          CALL NEW(PDS)
          CALL NEW(LI)
         S = LI
          CAR(LI) =INFTR
 3        IF(AF(CAR(LI)).NE.1) GOTO 1
           IF(CAR(LI).EQ.SOU)GOTO 2
 4        LI =CDR(LI)
          IF(LI.NE.0)GOTO 3
          CALL POPUP(LI,PDS)
          IF(LI.NE.0)GOTO 4
 6        CALL BACK(S)
          RETURN
 1       CALL PUSH(LI,PDS)
          LI=CAR(LI)
          GOTO 3
 2       CALL POPUP(I,PDS)
          IF(I.EQ.0) GOTO 6
          IF(CAR(CAR(I)).EQ. GOAL) GOTO 5
          GOTO 2
 5        CALL ERASE(PDS)
          CALL BACK(S)
          CROSS =1
          RETURN
          END
```

COMB

parameters: I1 and J where I1 and J are both set interpretations of feature complexes

operation:

The integer function COMB computes the extensional combination of two feature complexes and returns it as the value of COMB.
This is done by using the ADD subroutine which adds all atoms of a list to another list, if and only if the atoms are not already there.

code:

```
      INTEGER FUNCTION COMB (I1,J1)
      IMPLICIT INTEGER (A-W)
      LOGICAL*1 AF
      COMMON CAR(3000),CDR(3000),AF(3000)
      COMB = I1
      IF (J1.EQ.0) RETURN
      COMB = J1
      IF (I1.EQ.0) RETURN
      CALL NEW (COMB)
      C = COMB
      IC = C
      J = J1
    1 IF (J.EQ.0) GOTO 2
      I = I1
    4 IF (I.EQ.0) GOTO 3
      F = COPY (CAR(I))
      CALL ADD (CAR(J),F)
      CALL APPEND (C,F,C)
      I = CDR(I)
      GOTO 4
    3 J = CDR(J)
      GOTO 1
    2 COMB = CDR(IC)
      CALL BACK(IC)
      RETURN
      END
```

3.3.1.3. The implementation of the completion automata.

We use transition networks at various places in the whole system to control order restrictions. Let us now discuss the procedures that are able to consult the transition networks. These procedures are located in a subroutine called NETW.

(i) input:

Recall our conventions for representing transition networks in the form of list representations. A transition network is a list of quadruples: $\langle$a1,a2,a3,a4$\rangle$ where a1 is the start state of a transition, a2 is the resulting state, a3 is the condition for the transition to take place and a4 is the symbol associated with the transition.

a1 may be one state or a feature complex of states
a2 may be one state or a list of states
a3 is a feature complex containing features
a4 is one single element or a list of elements.

A transition network under the given conventions is the first main piece of input information (called NET).
The second main piece is a triple $\langle$CON, STAT,RES$\rangle$ where
CON denotes the condition for a transition to take place (CON is the extension of a feature complex)
STAT denotes a state (or a set of states)
RES denotes possibly a symbol associated with the transition.

The idea is that if CON is NIL, RES is the condition for a transition to take place, so we can perform transitions both on the basis of the condition itself and on the associated symbol.

(ii) output:

The output consists of two things:
(a) A value for NETW, the call name of the procedure with 0 or 1, denoting that no transition or at least one transition took place respectively, thus we can immediately check whether there was any result.

(b) A list of triples (called OUTP) ⟨b1,b2,b3⟩ with
b1 the resulting domain of the conditional feature complex
b2 a new state (or a set of new states)
b3 the symbol associated with the transition.

So we come to the following program:

NETW

parameters: CON, STAT, RES, OUTP, NET

operation:

The procedure is a straight forward list processing action computing the states and the features according to the specifications given. We introduce a flag (FL) to indicate whether the condition or the associated symbol will determine the transition. A pointer (INET) runs through the network. First a match is tried for the state, next a match for the condition of transitions.
If successful a new list (L) is created and attached to the OUTP(ut) list via an APPEND operation on the S-pointer.

code:

```
         INTEGER FUNCTION NETW(CON,STAT,RES,OUTP,NET,INT,FUNTRE)
         IMPLICIT INTEGER (A-W)
         LOGICAL*1 AF
         COMMON CAR(3000),CDR(3000),AF(3000)
         NETW =0
         FL=0
         IF(CON.EQ.0) FL=1
         IF(FL.EQ.1.AND.RES.EQ.0)RETURN
         INET = NET
         CALL NEW(OUTP)
         S = OUTP
         CALL NEW(IS)
         CAR(IS) = STAT
C CHECK WHETHER CONDITION IS SATISFIED
1        IF(INET.EQ.0) GOTO 10
         IRES =0
         IF(FL.EQ.1) GOTO 5
         IRES = MATCH (CAR(CDR(CDR(CAR(INET)))),CON,FUNTRE)
D       CALL PRLIST (IRES,1,6)
D       CALL PRLIST (CON,1,6)
         IF(IRES.EQ.0) GOTO 15
         GOTO 20
5        IF(RES.NE.CAR(CDR(CDR(CDR(CAR(INET)))))) GOTO 15
C CHECK WHETHER STATE IS SATISFIED
20      NSTAT = MATCH(CAR(CAR(INET)),IS,INT)
        CALL PRLIST (NSTAT,1,6)
        CALL PRLIST (IS,1,6)
         IF(NSTAT.EQ.0) GOTO 15
C ADD NEW TRIPLE TO OUTPUT
         CALL NEW(L)
         CALL APPEND (S,L,S)
         CAR(L) = IRES
         CALL APPEND (L,CAR(CDR(CAR(INET))),I)
         IF(CDR(CDR(CDR(CAR(INET)))).NE.0)
       * CALL APPEND (I,CAR(CDR(CDR(CDR(CAR(INET))))),I)
15       INET =CDR(INET)
         GOTO 1
C END
10       IF(CDR(OUTP).NE.0) GOTO 11
         CALL BACK(OUTP)
         RETURN
11        I = CDR(OUTP)
         CALL BACK(OUTP)
         NETW=1
         OUTP = I
         RETURN
         END
```

3.2.2. The main program

Let us now consider the main program of the parser.
It performs the following tasks:

(i) Initialization

This includes

(a) Internal initialization of the list structure memory and of the files on disk on which the dictionary is stored.

(b) Initialization of the variables which are needed in the parser. In particular we input all terms which will be common to the programming system and the user.

(c) As soon as the reader has given the language in which he wants to work, we also read the grammar, the syntactic networks and the relevant inference trees. After that the system is ready to consume an input sentence.

(ii) Preparation

Then a request is issued to the user for an input sentence. For each word in this sentence the system consults the dictionary and creates the initial particles according to the conventions we discussed in the previous chapter. The particles are organized as described earlier

(iii) Send to parser

When the initial particles have been made for a given input word, the program control shifts to the subroutine who actually controls the parsing process, namely the subroutine CONTR.

(iv) Send to semantic structurer

When all input words have been consumed in this way the program control shifts to the routines which extract functional structures, case structures and semantic structures from the particles which cover the complete input sentence.

code:

```
      IMPLICIT INTEGER (A-X)
      LOGICAL*1 TA
      LOGICAL*1 AF
      COMMON /IFREE/IFREE
      COMMON /VECT/ VECT(30),WORDS
      COMMON /INFTRE/SYNTRE,SEMTRE,FUNTRE
      COMMON/ADD/SYNAFT,VERBAL,CASE1
      COMMON /CODE/ LOCK,RULE,BEFORE,AFTER,TRUE,FALSE,UNDET,FUNCTW,
     * SYNNET,FRAME,OBJEC,UNMA,PREDIC
      COMMON /COD2/ MOD,QUAL,ADJU
      COMMON /INVES/ INVES,NSTATS,LD
      COMMON /LOG/ AND,OR,XOR,NOT
      COMMON /COMF/ COMF(30,10)
      COMMON /FIN/ FIN,TR
      COMMON /TA/ TA
      COMMON/PDS/ PDS,PDS2
      COMMON /V/ VERB
      COMMON CAR(3000),CDR(3000),AF(3000)
      CALL INIT
C READ SYMBOLS
      CALL ASSIGN (3,'WORD.DAT',0)
      CALL FDBSET(3,'UNKNOWN')
      DEFINE FILE 3(7993,17,U,IREC)
      CALL ASSIGN(4,'INFO.DAT',0)
      CALL FDBSET(4,'UNKNOWN')
      DEFINE FILE 4(5001,41,U,IREC)
      TR = 0
      NIL = 0
      CALL NEW(PDS)
      CALL NEW(PDS2)
      CODES = RLIST (0,I,2)
      ICODE = CODES
      MORE = CAR(ICODE)
      LOCK = CAR(CDR(ICODE))
      RULE = CAR(CDR(CDR(ICODE)))
      BEFORE = CAR(CDR(CDR(CDR(ICODE))))
      AFTER = CAR(CDR(CDR(CDR(CDR(ICODE)))))
      ICODE = CDR(CDR(CDR(CDR(CDR(ICODE)))))
      TRUE = CAR(ICODE)
      UNDET = CAR(CDR(ICODE))
      ADJU = CAR(CDR(CDR(ICODE)))
      FUNCTW = CAR(CDR(CDR(CDR(ICODE))))
      OBJEC = CAR(CDR(CDR(CDR(CDR(ICODE)))))
      ICODE = CDR(CDR(CDR(CDR(CDR(ICODE)))))
      FRAME = CAR(ICODE)
      SYNNET = CAR(CDR(ICODE))
      AND = CAR(CDR(CDR(ICODE)))
      OR = CAR(CDR(CDR(CDR(ICODE))))
      XOR = CAR(CDR(CDR(CDR(CDR(ICODE)))))
      ICODE = CDR(CDR(CDR(CDR(CDR(ICODE)))))
      NOT  = CAR(ICODE)
      PREDIC = CAR(CDR(ICODE))
      UNMA = CAR(CDR(CDR(ICODE)))
      MOD = CAR(CDR(CDR(CDR(ICODE))))
      QUAL = CAR(CDR(CDR(CDR(CDR(ICODE)))))
      ICODE = CDR(CDR(CDR(CDR(CDR(ICODE)))))
      FIN = CAR(ICODE)
```

```
           TRACE = CAR(CDR(ICODE))
           UNDO =  CAR(CDR(CDR(ICODE)))
           GRAMMA = CAR(CDR(CDR(CDR(ICODE))))
           SYNAFI= CAR(CDR(CDR(CDR(CDR(ICODE)))))
           ICODE = CDR(CDR(CDR(CDR(CDR(ICODE)))))
           SYNTR = CAR(ICODE)
           SEMTR= CAR(CDR(ICODE))
           FUNCTR= CAR(CDR(CDR(ICODE)))
           VERBAL = CAR(CDR(CDR(CDR(ICODE))))
           FEAT = CAR(CDR(CDR(CDR(CDR(ICODE)))))
           ICODE = CDR(CDR(CDR(CDR(CDR(ICODE)))))
           ARG = CAR(ICODE)
           PRS = CAR(CDR(ICODE))
           SEMSTR=CAR(CDR(CDR(ICODE)))
           HYPL = PLIST(0,I,2)
           HYPP = HYPL
           VERB = PLIST(0,I,2)
           OLIST = PLIST (0,I,2)
           WRITE(6,1000)
     1000  FORMAT (1X/1X,'WELCOME TO THE PARSING SYSTEM ')
           WRITE(6,1001)
     1001  FORMAT (1X/1X, 'SPECIFY THE LANGUAGE')
     112   READ(1,11) TA
     11    FORMAT (A1)
     D     WRITE (6,113) TA
     D113  FORMAT (1X, 'INPUT LANGUAGE :',A1)
     C READ THE GRAMMAR
           J = 0
           GRAM = SEARCH(GRAMMA)
           IG = GRAM
      12   I = CAR(GRAM)
           L = 0
           J = J+1
           CALL PROP(CAR(I),RULE,J)
     13     L = L+1
           K = CAR(I)
           IF(AF(K).NE.1) K = COPY(K)
           COMF(J,L) = K
           IF(CDR(I).EQ.0) GOTO 15
           I = CDR(I)
           GOTO 13
     15    IF(CDR(GRAM).EQ.0) GOTO 20
           GRAM = CDR(GRAM)
           GOTO 12
     20    CALL ERASE(IG)
     C READ THE NETWORKS
           NETS = SEARCH(BEFORE)
           IF (NETS.EQ.0) GOTO 26
           LAB = BEFORE
     23    IN = NETS
     21    CALL PROP(CAR(CAR(IN)),LAB,CDR(CAR(IN)))
           IN = CDR(IN)
           IF(IN.NE.0) GOTO 21
           IF (LAB.EQ.AFTER) GOTO 27
     26    NETS = SEARCH(AFTER)
           LAB = AFTER
           IF (NETS.NE.0) GOTO 23
```

```
      C READ INFERENCE TREES
      27     SYNTRE = SEARCH (SYNTR)
             SEMTRE = SEARCH (SEMTR)
             FUNTRE = SEARCH (FUNCTR)
      25     CALL NEW(NSTATS)
             CALL NEW(LO)
             CALL NEW (NSTATS)
      30     WRITE(6,1002)
      1002   FORMAT (1X, 'GIVE INPUT SENTENCE')
             WORDS = 0
      C SENTENCE COMES IN
             INP = RLIST(0,I,1)
             IF(INP.EQ.0) GOTO 550
             I = INP
             IF(INP.EQ.TRACE) TR = 1
             IF(INP.EQ.UNDO) TR = 0
             IF((INP.EQ.TRACE).OR.(INP.EQ.UNDO)) GOTO 30
             INPP = INP
             IF(INPP.EQ.0) GOTO 550
      35     J = SEARCH(CAR(I))
             IF(CDR(I).EQ.0) GOTO 40
             I = CDR(I)
             GOTO 35
      40     CONTINUE
             WRITE(6,1003)
      1003   FORMAT (1X/1X,'IN:')
             CALL PRLIST(INP,1,6)
             IF(TR.EQ.1) WRITE(6,1004)
      1004   FORMAT (1X, 'CONFIGURATIONS IN THE STATESPACE:')
      D45    TR = 1
      C TAKE NEW WORD
      50     WORD = CAR(INPP)
             WORDS = WORDS +1
      D      WRITE(6,102)
      D102   FORMAT (1X/1X)
      D      CALL PRLIST (WORD,16,6)
      D      WRITE(6,1005) WORDS
      D1005 FORMAT (1H+, 'WORD NR :',I3)
             CALL GET(WORD,-1,IMORE)
             IF(IMORE.EQ.0) GOTO 550
      C
      C CONSTRUCT INITIAL STRUCTURE
      C
      D      WRITE(6,104)
      D104   FORMAT (1X, '.I. INITIAL PARTICLES :')
             CALL NEW(K)
             WLIST = K
             CAR(K) = WORD
      1      CALL NEW(L)
             CALL APPEND (K,L,K)
             CAR(L) = CAR(HYPL)
             ON = L
             HYP = CAR(HYPL)
             CALL PROP(WORD,HYP,L)
             HYPL = CDR(HYPL)
             FLAG = 0
      C      FOR EACH LEXICAL INFORMATION LINE CONSTRUCT PARTICLE
```

```
            FUNC = CAR(CAR(IMORF))
            IFUN = 0
            IF(AF(FUNC).EQ.1) GOTO 4
            IFUN = FUNC
     2      FUNC = CAR(IFUN)
            IFUN = CDR(IFUN)
     32     IF (FLAG.EQ.1) GOTO 3
            FLAG = 1
            GOTO 4
     3      CALL NEW(L)
            CALL APPEND (K,L,K)
            CAR(L) = CAR(HYPL)
            ON = L
            HYP = CAR(HYPL)
            CALL PROP(WORD,HYP,L)
            HYPL = CDR(HYPL)
     4      CALL APPEND (L,CAR(IMORF),L)
            CALL NEW(F)
            CALL APPEND (L,F,L)
            IF(WORDS.EQ.1) GOTO 5
            CALL PUSH(F,INVES)
     5      CALL NEW(J)
            CAR(F) = J
            CALL APPEND (F,WORDS-1,F)
            CDR(F) = ON
            CALL GET(FUNC,RULE,IR)
            IF (IR.EQ.0) GOTO 550
            NNET = 0
            ANET =0
            CALL GET (FUNC,BEFORE,NNET)
            CALL GET (FUNC,AFTER,ANET)
     C (A)  WORD
            IF(WORDS.NE.1.AND.NNET.NE.0) CAR(J) = CAR(NNET)
            CALL APPEND (J,WORD,J)
     C (B)  INFORMATION SEQUENCE
            CALL NEW(I)
            CALL APPEND (J,I,J)
     C (1)  HYPOTHESIS
            CAR(I) = HYP
     C (2)  FUNCTION NAME
            CALL APPEND (I,FUNC,I)
     C(3) STATE OF FUNCTION FOR AFTER TRANSITIONS
            CALL APPEND (I,0,J)
            IF (ANET.NE.0) CAR(J) = CAR(ANET)
            I = J
     C (4)  STATE IN CASE NETWORK (UNKNOWN YET)
            CALL APPEND (I,0,I)
     C ADJUNCTS
            IF(COMF(IR,2).EQ.OBJEC) GOTO 6
     C(5) EXTERNAL FEATURE COMPLEX * QUAL-MOD-UNDET CHARACTERISTIC
            I4 = CDR(CDR(CDR(CDR(CAR(IMORF)))))
            CALL APPEND (I,V,I)
            CALL APPEND (I,COMF(IR,9),J)
            IF (I4.EQ.0) GOTO 9
            I4 = CAR(I4)
            IF (I4.EQ.0) GOTO 9
            IF (AF(I4).EQ.1.OR.CAR(I4).EQ.NOT.OR.CAR(I4).EQ.
```

```
      :     AND.OR.CAR(I4).EQ.OR.OR.CAR(I4).EQ.XOR) GOTO 9
            CAR(I) = EXT(CAR(COR(I4)))
            GOTO 9
C OBJECTS
C  (5) SYNT FEAT COMPLEX
 6          J = EXT(CAR(CDR(CDR(CDR(CDR(CAR(IMORF)))))))
            CALL APPEND (I,J,I)
C  (6) SEM FEAT COMPLEX
            CASE = CAR(CDR(CDR(CDR(CAR(IMORF)))))
            J = SEARCH(CAR(CDR(CAR(IMORF))))
 7          IF(CAR(CAR(J)).EQ.CASE) GOTO 8
            J = CDR(J)
            IF(J.NE.0) GOTO 7
            WRITE(6,1006)
 1006       FORMAT (1X, 'MISSING CASE IN FRAME ')
            GOTO 94
 8          CALL APPEND (I,EXT(CAR(CDR(CAR(J)))),I)
C(7) CASE (UNKNOWN YET EXCEPT FOR ADJUNCTIVE OBJECTS)
            CALL APPEND (I,0,I)
C
 9          IF(TR.EQ.1) CALL PRLIST(CAR(CAR(L)),1,6)
            IF(IFUN.NE.0) GOTO 2
            IMORF = CDR(IMORF)
            IF(IMORF.NE.NIL) GOTO 1
            VECT(WORDS) = CDR(WLIST)
            IF(WORDS.EQ.1) GOTO 111
D           WRITE(6,557)
D557        FORMAT (1X,'.II. MERGING')
C
C START PARSING
C
            CALL CONTR
 111        IF(CDR(INPP).EQ.0) GOTO 10
            INPP = CDR(INPP)
            GOTO 50
C
C COMPUTE SEMANTIC STRUCTUES
 10         FINL = VECT(WORDS)
            HYPL = HYPP
            T = 0
            WRITE (6,440)
 440        FORMAT (1X/1X,'FUNCTIONAL AND CASE STRUCTURES :')
            CALL CLOSE(4)
 93         HYP = CAR(FINL)
            FEAT = CAR(CDR(HYP))
            CONF = CDR(CDR(HYP))
 92         IF(CAR(CDR(CAR(CONF))).NE.0) GOTO 90
            I =  CDR(CAR(CAR(CONF)))
            CALL FUN(I)
            IF(I.EQ.0) GOTO 90
            T = T+1
            CALL CAS(I)
 90         CONF = CDR(CONF)
            IF(CONF.EQ.0) GOTO 91
            GOTO 92
 91         FINL = CDR(FINL)
            IF(FINL.NE.0) GOTO 93
            IF (T.EQ.0) WRITE (6,556)
 556        FORMAT (1X,'NO STRUCTURE FOR GIVEN INPUT')
 94         CONTINUE
D           TR = 0
            WRITE(6,555) 3000-IFREE
 555        FORMAT (1X/1X 'MEMORY CELLS LEFT:'I4)
            CALL CLOSE(4)
            CALL ASSIGN(4,'INFO.DAT',0)
            CALL FDBSET(4,'UNKNOWN')
            DEFINE FILE 4(5001,41,U,IREC)
            GOTO 30
 550        CONTINUE
            END
```

3.2.3 The general control structure

CONTR

parameters: none

operation:

The subroutine CONTR is the actual control program of the parser. It takes two configurations and sends them to the subroutine LR which performs the linguistic processes (computation of parsing predicates and creation of new particles).

The subroutine operates on the basis of a tasklist and a task is a configuration in a particle that is to be investigated. The main program places the initial tasks on this tasklist (called INVES) and whenever new particles have been made (by LR) they are placed on the tasklist to see whether new combinations are possible.

CONTR takes one configuration from the tasklist. According to the principle that a particle can only merge with particles bordering on its domain, CONTR scans all particles depending on each hypothesis node of the word immediately before the domain of a given particle. When these particles are not locked, they are made subject to the linguistic processor. Moreover a pointer is provided to which part of the particle the other particle is supposed to be related. If the particle has been processed, we go back to the tasklist to see if there are still other particles.

The final part of CONTR contains the procedure to attach configurations to the relevant hypothesis node and to 'lock' a particle if told so by the linguistic processor.

code:

```
          SUBROUTINE CONTR
          IMPLICIT INTEGER (A-W)
          LOGICAL*1 ALF(10)
          LOGICAL*1 AF
          COMMON CAR(3000),CDR(3000),AF(3000)
          COMMON /COMF/ COMF(30,10)
          COMMON /CODE/ LOCK,RULE,BEFORE,AFTER,TRUE,FALSE,UNDET,FUNCTW,
         * SYNNET,FRAME,OBJEC,UNMA,PREDIC
          COMMON/INVES/ INVES,NSTATS,LO
          COMMON /VECT/ VECT(30),WORDS
          COMMON /V/ VERB
          COMMON/POS/ POS,POS2
          COMMON /IFREE/ IFREE
          DATA ALF/'A','B','C','D','E','F','G','H','I','J'/
          S = NSTATS
    D     A = 0
    C TAKE TASK FROM TASKLIST
    1     IF(CAR(INVES).EQ.0) GOTO 10
          CALL POPUP(CONF,INVES)
          STRUCT = CAR(CONF)
          OWOR = CAR(CDR(CONF))
    D     A = A+1
    D     WRITE(6,101) ALF(A)
    D101  FORMAT (1X, '(',A1,')')
    D     WRITE(6,100)
    D100  FORMAT (1X,'**** TRY TO EXPAND CONFIGURATION :')
    D     CALL PRLIST(STRUCT,5,6)
          OHYPL = VECT(OWOR)
    D     WRITE(6,102) OWOR
    D102  FORMAT ('** BY COMBINING IT WITH CONFIG OF WORD NR.',I3)
    D     T1 = 0
    C GET PARTICLES BORDERING ON INVESTIGATED CONFIG
    2     OHYP = CAR(OHYPL)
    D     T1 = T1 +1
    D     CALL PRLIST(CAR(OHYP),22,6)
    D     WRITE(6,107)       T1
    D107  FORMAT (1H+,I2,'. FOR HYPOTHESIS :')
    D     T2 = 0
          OCONFS = CDR(CDR(OHYP))
    203   OCONF = CAR(OCONFS)
          IF(CAR(CAR(OCONF)).EQ.LOCK) GOTO 199
          I = CAR(CDR(CAR(CDR(CDR(CAR(OCONF))))))
          J = CAR(CDR(CAR(CDR(CDR(CAR(CONF))))))       )
          IF(I.EQ.VERB.AND.J.EQ.VERB) GOTO 199
    D     T2 = T2 +1
    D     WRITE(6,103) T1,T2
    D103  FORMAT (3X ,I2,'.',I2,'.',' CONFIGURATION :')
    D     CALL PRLIST(CAR(OCONF),4,6)
    D     T3 = 0
          IF(CAR(CAR(OCONF)).EQ.PREDIC) GOTO 204
    C CALL LINGUISTIC PROCESSOR FOR LEFT TO RIGHT COMBINATION
    D     WRITE(6,104)
    D104  FORMAT (5X '=> FROM LEFT TO RIGHT')
          CALL LR(CONF,OCONF,0,CDR(CAR(CONF)))
    204   CONTINUE
    C CALL LINGUISTIC PROCESSOR FOR RIGHT TO LEFT COMBINATION
    C FOR EACH "RIGHTMOST NODE " IN THE STRUCTURE
```

```
D          WRITE(6,105)
D105       FORMAT (5X,'<= FROM RIGHT TO LEFT')
           I = CDR(CAR(OCONF))
           POIN = I
201        I = CDR(I)
200        IF(CDR(I).NE.0) GOTO 196
D          CALL PRLIST(CAR(POIN),29,6)
D          T3 = T3 +1
D          WRITE(6,106) T1,T2,T3
D106       FORMAT (1H+,7X,I2,'.',I2,'.',I2,'. FOR WORD :')
           CALL LR(OCONF,CONF,1,POIN)
197        IF(CAR(PDS).EQ.0) GOTO 199
           CALL POPUP(I,PDS)
           CALL POPUP(POIN,PDS2)
           IF(I.EQ.0) GOTO 199
           GOTO 200
196        IF(CAR(CDR(I)).EQ.0) GOTO 197
           I = CDR(I)
           CALL PUSH(I,PDS)
           CALL PUSH(POIN,PDS2)
           I = CAR(I)
           POIN = I
           GOTO 201
199        IF(CDR(OCONFS).EQ.0) GOTO 202
           OCONFS = CDR(OCONFS)
           GOTO 203
202        CONTINUE
3          IF(CDR(OHYPL).EQ.0) GOTO 1
           OHYPL = CDR(OHYPL)
           GOTO 2
C ATTACH RESULTING PARTICLES AND LOCK
10          NSTATS = S
12         IF(CAR(NSTATS).EQ.0) GOTO 13
           CALL POPUP(J,NSTATS)
           CONF = J
           NHYP = CDR(CDR(CONF))
           I = CDR(NHYP)
11         I = CDR(I)
           IF(CDR(I).NE.0) GOTO 11
           CALL APPEND (I,J,I)
           GOTO 12
13         IF(CAR(LO).EQ.0) RETURN
           CALL POPUP(I,LO)
           CAR(CAR(I)) = LOCK
           GOTO 13
           END
```

3.2.4. The linguistic processor.

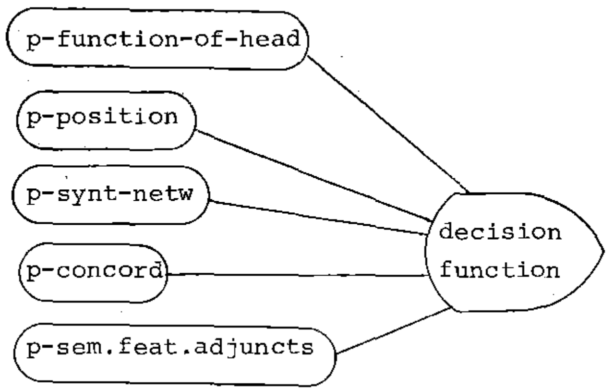
LR

parameters : none

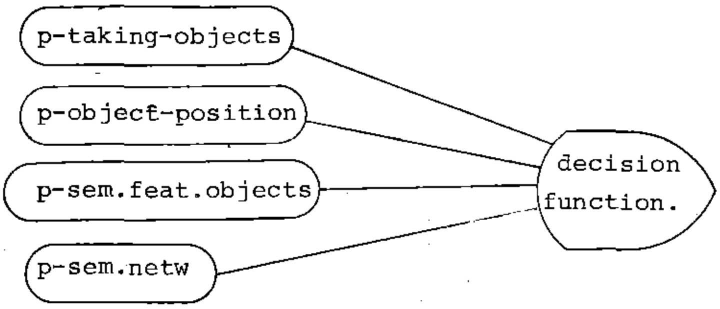
operation:
This subroutine performs two main tasks:
(i) The computation of the parsing predicates, and
(ii) The construction of new configurations when merging two particles. This first task is further subdivided in two main areas (a) the execution of the parsing predicates for adjuncts and functionwords and (b) the execution of the parsing predicates for objects.
After the necessary preparation (such as getting the relevant information pointers into the lexicon and to the syntactic rules) we start computing the parsing predicates.

When considering the whole set of parsing predicates and in particular and in particular the domains for which they are defined we come to the following scheme:

(i) predicates for adjuncts and function words:



(ii) predicates for objects:

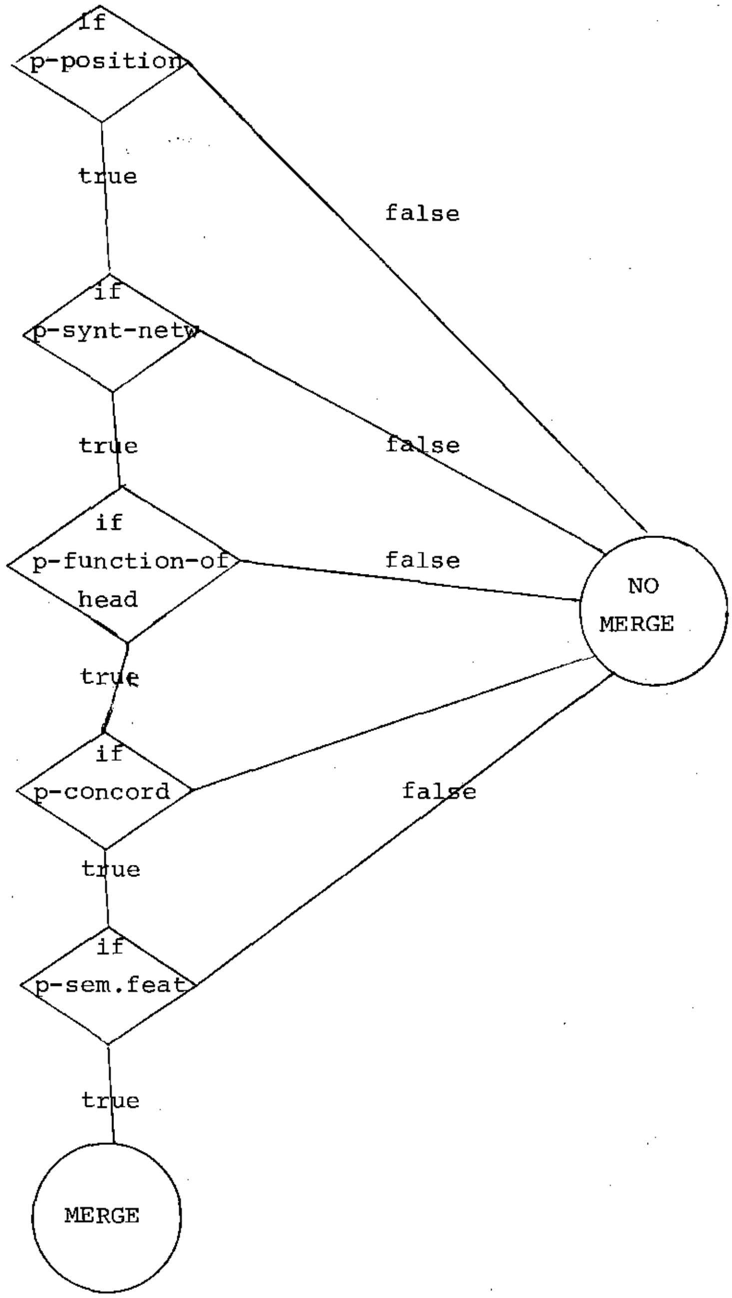


For the investigation and development of the system at the current state of knowledge and on computers which do not allow parallel computation (except by sequential simulation) we decided to implement a sequential instead of a perceptron like control structure, that means: we apply each predicate after the other one and as soon as one predicate fails we abandon the idea of merging. We stress that this method will fail to account for the various points which were given in favour of a perceptron control. Nevertheless the sequential control structure proves to be extremely useful in research for the grammar, i.e. the strict contents of linguistic knowledge; we want to know precisely how far the linguistic information goes and where it rejects.

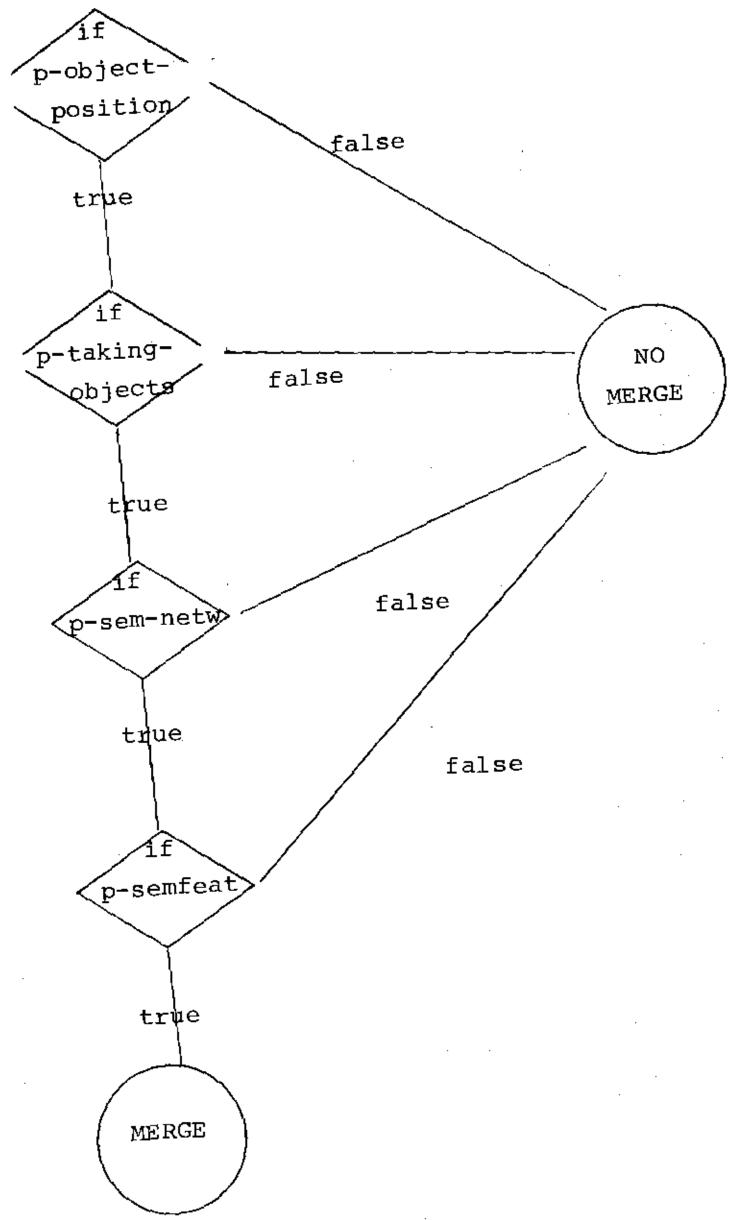
We found out that the following flow of control is most efficient, that means the fastest rejection of a possible merging by as little as possible of computation.

(i) for adjuncts/functionwords:

parser implementation:



for objects:



if
p-object-
position
false
true
if
p-taking-
objects
false
NO
MERGE
true
if
p-sem-netw
false
true
false
if
p-semfeat
true
MERGE

A deviation occurs for objective adjuncts which follow the flow of control of adjuncts except that instead of the p-position predicate comes the p-object-position predicate.
Similarly for adjunctive objects, they follow the control structure of objects except that instead of the p-object-position predicate, the p-position predicate is used.

Now we give some comments on the computation of the predicates themselves. In principle each time a predicate is true, a message is produced,and when it is false another message is produced and we return back to the calling routine CONTR.

(1) Networks
We prepare the call to NETW by (i) getting the networks and (ii) constructing a special list format for the function which acts as condition of the transition.
Then we call the routine NETW which performs a transition if allowed by the data, and filter out the result in the main routine.

(2) Function-of-head/position

When the networks have been unsuccessful we check on the basis of the grammar itself whether the function-of-head/ or taking-objects rule and the position or object-position rule respectively applies. If successful we proceed, else the linguistic processor returns control to CONTR.

From now on the parsing predicates computation is performed in two separate parts:

(A) ADJUNCTS and FUNCTIONWORDS

(3) Syntactic features

If the grammar prescribes agreement we fetch the relevant feature complexes and send them to the MATCH routines. If the result is false, control shifts back to the CONTR program. Moreover if the grammar prescribes sending through features to the head, the relevant preparation is performed and the features are sent-through by means of the subroutine COMB.

(4) Semantic features

Finally we do the semantic features test for adjuncts which is mainly located in the subroutine FRAMES. A complication arises in getting the relevant information in certain verbal constructions where the semantic features test is performed on the subject of the verb.
If the FRAMES test is positive we go to the second main part of the LR subroutine: the construction of new information structures.

(B) OBJECTS

(1) Surface case signals

For objects we perform after the order/relations environment tests the tests of surface case signals. To this purpose we compute the relevant surface case networks by means of viewpoint andfunction . Then we call the NETW program that consults the semantic networks and delivers a (possibly empty) list of triples syntactic features/states/cases.

(2) Semantic features

Finally we compute the semantic features associated with the case slots found by the surface case processing and perform a match with the sematnic features associated with that word. If there is at least one case for which a match is successful we construct new configurations.

II. New configurations

The construction of new configurations is a complex book keeping task.

(1) Changes in the subordinate
First of all we make a copy of the configuration of the subordinate and change the information resulting as a side effect from the execution of the parsing predicates.

(2) Particle superstructure
Then we construct a copy of the configuration of the head and attach the old configuration to the new one. This is a quite complex process. Not only do we need to add information about the domain, e.g., but we also have to look into the structure of the head configuration if the subordinate is not attached on the topnode. This is done by a subroutine NPOINT (to be discussed soon).

(3) Changes in head configuration
Finally we make the changes in the information of the head configuration as specified earlier. A special procedure comes then into operation for verbs, in particular we reverse the usual head-subordinate structure. This turns out to lead to a more efficient semantic structuring process and to a more efficient representation for the rest of the parsing process.

code:

```
      SUBROUTINE LR(NCONF,OCONF,F,POIN)
      IMPLICIT INTEGER (A-X)
      LOGICAL*1 AF
      COMMON CAR(3000),CDR(3000),AF(3000)
      COMMON/LOG/AND,OR,XOR,NOT
       COMMON /INFTRE/SYNTRE,SEMTRE,FUNTRE
      COMMON /COMF/ COMF(30,10)
      COMMON /CODE/ LOCK,RULE,BEFORE,AFTER,TRUE,FALSE,UNDET,FUNCTW,
     * SYNNET,FRAME,OBJEC,UNMA,PREDIC
      COMMON/INVES/ INVES,NSTATS,LO
      COMMON /FIN/ FIN,TR
      COMMON /IFREE/IFREE
      COMMON /COD2/ MOD,QUAL,ADJU
      COMMON/ADD/ SYNAFT,VERBAL,CASE1
C INITIALIZE CHANGE INDICATORS
      OSEM = 0
      ANEWS = 0
      RES = 0
      OSYN = 0
      IN = 0
      OUTP = 0
      ICASE = 0
      NSEM = 0
      NSYN = 0
      CHAR = 0
      NEWS = 0
      CASEST = 0
      STYP = 0
      OU = 0
      NRES = 0
D     OU = 1
C GET RELEVANT INFORMATION POINTERS
      NSTRUC = CDR(CAR(NCONF))
      STRUCT = CAR(NCONF)
      OSTRUC = CDR(CAR(OCONF))
      CALL GET (CAR(OSTRUC),CAR(CAR(CDR(OSTRUC))),OHYP)
      CALL GET(CAR(POIN),CAR(CAR(CDR(POIN))),NHYP)
C GET LEXICON INFORMATION (O/N-FEAT)
      OFEAT = CAR(CDR(OHYP))
      NFEAT = CAR(CDR(NHYP))
C GET INFORMATION SEQUENCE (O/I-INF)
       OINF = CAR(CDR(OSTRUC))
      NINF = CAR(CDR(POIN))
C GET FUNCTION (O/N-FUNC)
      OFUNC = CAR(CDR(OINF))
      NFUNC = CAR(CDR(NINF))
C GET SYNTACTIC RULE (O/I-RULE)
      CALL GET(NFUNC,RULE,NRULE)
      CALL GET(OFUNC,RULE,ORULE)
C (1) NETWORKS
C (A) GET NETWORK
 1        IF(F.EQ.0) CALL GET (NFUNC,BEFORE,NNET)
      IF(F.EQ.1) CALL GET (NFUNC,AFTER,NNET)
      IF(NNET.EQ.0) GOTO 2
C (B) GET STATE
      IF(F.EQ.0) NSTATE =CAR(CAR(NCONF))
      IF(F.EQ.1)NSTATE = CAR(CDR(CDR(NINF)))
```

```
      IF(NSTATE.EQ.0) NSTATE = CAR(NNET)
      INFTR = CAR(CDR(NNET))
      NNET = CAR(CDR(CDR(NNET)))
C(C) PREPARE INPUT FOR NETW
      CALL NEW(COND)
      CALL NEW(I)
      CAR(I) = OFUNC
      CAR(COND) = I
      L = 0
      J = NSTATE
C(D) CONSULT
      I = NETW(COND,NSTATE,L,K,NNET,INFTR,FUNTRE)
      CALL ERASE (COND)
      IF(I.EQ.0) GOTO 2
C(E) FILTER
      CALL NEW(NEWS)
      L = NEWS
      I = K
11    IF(I.EQ.0) GOTO 12
      CALL ADD(CAR(CDR(CAR(I))),NEWS)
      I = CDR(I)
      GOTO 11
12    NEWS = CDR(NEWS)
      CALL BACK(L)
D     CALL PRLIST (J,35,6)
D     WRITE (6,100)
D100  FORMAT (1H+,7X,'SUCCESSFUL TRANSITION FROM')
D     CALL PRLIST (NEWS,31,6)
D     WRITE(6,300)
300   FORMAT (1H+,7X,'TO THE NEW STATE(S) :')
      IF (F.EQ.1) ANEWS = NEWS
      IF (F.EQ.1) NEWS = 0
      GOTO 3
C(2) FUNCTION OF HEAD / POSITION
2     POS = 0
      IF(COMF(ORULE,3).EQ.OBJEC) POS = COMF(NRULE,6)
      IF(COMF(ORULE,3).NE.OBJEC) POS = COMF(ORULE,5)
      IF(POS.EQ.0) GOTO 1001
      IF(F.EQ.0.AND.POS.EQ.AFTER) GOTO 1001
      IF(F.EQ.1.AND.POS.EQ.BEFORE) GOTO 1001
      CALL NEW(COND)
      CALL NEW(I)
      CAR(I) = NFUNC
      CAR(COND) = I
      I=MATCH(COMF(ORULE,4),COND,FUNTRE)
      IF(I.EQ.0) GOTO 1001
D     WRITE(6,101)
D101  FORMAT (8X, 'SUCCESSFUL ORDER AND RELATIONS ENVIRONMENT TESTS')
C(3) SYNT FEATURES
3     IF(COMF(ORULE,3).EQ.OBJEC) GOTO 6
      IF(COMF(ORULE,7).NE.TRUE) GOTO 35
C(I) GET FEATURES
      NDOM = CAR(CDR(CDR(CDR(CDR(NINF)))))
      OFEAS = CAR(CDR(CDR(CDR(CDR(OFEAT)))))
      IF (AF(OFEAS).EQ.1) GOTO 31
      IF (CAR(OFEAS).EQ.AND.OR.CAR(OFEAS).EQ.OR.OR.CAR(OFEAS)
     * .EQ.XOR.OR.CAR(OFEAS).EQ.NOT)  GOTO 31
```

```
0110          DFEAS = CAR(DFEAS)
0111    31    CONTINUE
        C(II) MATCHING
        D     WRITE (6,103)
        D103  FORMAT (8X,'MATCH THE FOLLOWING FEATURE COMPLEXES:')
        D     CALL PRLIST (DFEAS,8,6)
        D     CALL PRLIST (NDOM,8,6)
0112          RES = MATCH (DFEAS, NDOM,SYNTRE)
0113          IF (RES.EQ.0) GOTO 1002
0115    44    CONTINUE
        D     WRITE (6,102)
        D102  FORMAT (8X,'RESULTING DOMAIN:')
        D     CALL PRLIST (RES,8,6)
0116          NSYN = RES
        C (III) SEND-THROUGH
0117    35    IF(COMF(ORULE,8).NE.TRUE) GOTO 4
0119          IF (RES.NE.0) RES = COPY(RES)
0121          IF (RES.EQ.0) RES = CAR(CDR(CDR(CDR(CDR(NINF)))))
0123          NSYN = COMB (EXT(CAR(CDR(CDR(CDR(CDR(CDR(OFEAT))))))),RES)
        D     WRITE (6,106)
        D106  FORMAT (8X,'NEW FEATURE COMPLEX:')
        D     CALL PRLIST (NSYN,8,6)
        C (4) SEMANTIC FEATURES TEST
0124    4     IF (COMF(ORULE,9).EQ.0) GOTO 5
        C(I) SEARCH INFORMATION SEQUENCES
0126          INFEAT = NFEAT
0127          ININF = NINF
0128          INRULE = NRULE
0129          IF (OFUNC.NE.VERBAL) GOTO 41
0131          SUBJ = CAR(CDR(CDR(CDR(STRUCT))))
0132          IF (CAR(CDR(CDR(NINF))).NE.FIN) GOTO 1003
0134          CALL GET (CAR(SUBJ),CAR(CAR(CDR(SUBJ))),INHYP)
0135          INFEAT = CAR(CDR(INHYP))
0136          ININF = CAR(CDR(SUBJ))
0137          CALL GET (CAR(INFEAT),RULE,INRULE)
0138    41    I = 0
0139          IF (COMF(INRULE,2).EQ.OBJEC) I =
             : CAR(CDR(CDR(CDR(CDR(CDR(ININF))))))
0141          STYP = COMF(ORULE,9)
0142          NRES = FRAMES (INFEAT,OFEAT,STYP,I)
0143          IF (NRES.EQ.0) GOTO 1003
        D     WRITE (6,107)
        D107  FORMAT (8X,'SEMANTIC FEATURES MATCH SUCCESSFUL, DOMAIN :')
        D     CALL PRLIST(NRES,8,6)
0145          IN = 1
0146          GOTO 5
        C
        C(B) OBJECT
        C
        C(1) SEMANTIC NETWORKS FOR SURFACE CASE SIGNALS
0147    6     ROLES = SEARCH (CAR(CDR(NFEAT)))
0148          NROLE = CAR(CDR(CDR(CDR(NFEAT))))
0149          CALL NEW (NFUNS)
0150          CALL NEW (I)
0151          CAR(NFUNS) = I
0152          CAR (I) = NFUNC
0153    61    IF (CAR(CAR(ROLES)).EQ.NROLE) GOTO 62
```

```
            ROLES = CDR(ROLES)
            IF (ROLES.EQ.NROLE) GOTO 62
            IF (ROLES.EQ.0) GOTO 1005
            GOTO 61
      62    ASSO = CDR(CDR(CAR(ROLES)))
            IF (ASSO.EQ.0) GOTO 1005
      63    IF (MATCH(CAR(CAR(ASSO)),NFUNS,FUNTRE).NE.0) GOTO 64
            ASSO = CDR(ASSO)
            IF (ASSO.EQ.0) GOTO 1005
            GOTO 63
      64    NNET = CDR(CAR(ASSO))
            FEATS = CAR(CDR(CDR(CDR(CDR(OINF)))))
      D     WRITE (6,109)
      D109  FORMAT (8X, 'CONSULT CASE FRAMES WITH SYNT FEATURES :')
      D     CALL PRLIST (FEATS,8,6)
            CASEST = CAR(CDR(CDR(CDR(NINF))))
            IF (CASEST.EQ.0) CASEST = CAR(NNET)
            IF (CASEST.EQ.0) GOTO 1006
            S = NETW(FEATS,CASEST,0,OUTP,CAR(CDR(CDR(NNET)))
           1     ,CAR(CDR(NNET)),SYNTRE)
            IF (OUTP.EQ.0) GOTO 1006
      D     WRITE (6,111)
      D111  FORMAT (8X,'SUCCESSFUL TRANSITION IN SEMANTIC NETWORKS'
           1     /8X,'RESULTING TRIPLES (FEATURES * STATE * CASE)')
***** C
      D     CALL PRLIST (OUTP,8,6)
      C SEMANTIC FEATURES
      D     WRITE (6,114)
      D114  FORMAT (8X,'MATCH THE FOLLOWING SEMANTIC FEATURES  ')
            SEMF = CAR(CDR(CDR(CDR(CDR(CDR(OINF))))))
      D     CALL PRLIST (SEMF,8,6)
      D     WRITE (6,112)
      D112  FORMAT (8X,'WITH FEATURES OF RESP. CASES ')
            I = OUTP
            CALL NEW (OUTP)
            IL = OUTP
      65    ICASE = CAR(CDR(CDR(CAR(I))))
            CALL PRLIST (ICASE,8,6)
            OROLES = SEARCH (CAR(CDR(NFEAT)))
      69    IF (CAR(CAR(OROLES)).EQ.ICASE) GOTO 66
            OROLES = CDR(OROLES)
            IF (OROLES.EQ.0) GOTO 1005
            GOTO 69
      66    OSEMF = CAR(CDR(CAR(OROLES)))
      D     CALL PRLIST (OSEMF,8,6)
            J = MATCH(OSEMF,SEMF,SEMTRE)
            IF (J.EQ.0) GOTO 68
      D     WRITE (6,116)
      D116  FORMAT (8X,'SEM FEATURES MATCH SUCCESSFUL')
            CALL APPEND (CDR(CDR(CAR(I))),J,L)
            CALL APPEND (OUTP,CAR(I),OUTP)
            IN = IN +1
            GOTO 67
      68    CONTINUE
      D     WRITE (6,117)
      D117  FORMAT (8X,'NO SEM FEATURES MATCH')
      67    I = CDR(I)
```

```
      IF (I.NE.0) GOTO 65
      IF (CDR(IL).EQ.0) GOTO 1007
      OUTP = CDR(IL)
      CALL BACK(IL)
      IF (I.EQ.0.AND.IN.EQ.0) GOTO 1007
C
S     CONTINUE
D     WRITE(6,105)
D105  FORMAT (1X, '>>>> ALL TESTS SUCCESSFUL, NEW CONFIGURATION :')
      DO 58 IO = 1,IN
      IF (OUTP.EQ.0) GOTO 59
      OSYNTF = CAR(CAR(OUTP))
      NSEM = CAR(CDR(CDR(CDR(CAR(OUTP)))))
      ICASE = CAR(CDR(CDR(CAR(OUTP))))
      CASEST = CAR(CDR(CAR(OUTP)))
      OUTP = CDR(OUTP)
C(1) CHANGES IN SUBORDINATE CONFIGURATION
59      ONEW = COPY (USTRUC)
      FES = CDR(CAR(CDR(ONEW)))
      IF(COMF(ORULE,2).NE.OBJEC) GOTO 193
C (A) FOR OBJECTS
      I3 = CDR(CDR(CDR(FES)))
C(I) SYNT FEAT
      IF(CAR(I3).NE.0) CALL ERASE(CAR(I3))
      CAR (I3) = OSYNTF
C(II) SEM FEAT
      IF(CAR(CDR(I3)).NE.0) CALL ERASE(CAR(CDR(I3)))
      CAR(CDR(I3)) = NSEM
C(III) CASE
      CAR(CDR(CDR(I3))) = ICASE
      GOTO 194
C (B) ADJUNCTS
193   IF(OFUNC.EQ.VERBAL)CAR (CDR(CDR(CDR(FES ))))=
     *         NSYN
      IF (OFUNC.EQ.SYNNET) CAR(CDR(FES)) = FIN
      IF(STYP.NE.0)CAR(CDR(CDR(CDR(CDR(FES)))))= STYP
C(2) CONSTRUCT PIRTICLE SUPERSTRUCTURE
194   CALL NEW(NSTATE)
      NSTRUC = COPY(CAR(NCONF))
      CAR(NSTATE) = NSTRUC
C RANGE
      IF(F.EQ.1) GOTO 201
C FOR DIRECTION LEFT TO RIGHT
200   CALL APPEND (NSTATE,CAR(CDR(OCONF)),J)
      CDR(J) = CDR(CDR(NCONF))
      CALL PUSH(NCONF,LO)
      GOTO 207
C FOR DIRECTION RIGHT TO LEFT
201   CALL APPEND (NSTATE,CAR(CDR(NCONF)),J)
      CDR(J) = CDR(CDR(OCONF))
      CALL PUSH(OCONF,LO)
C PUSH ON NSTATS,INVES,LOCK
207   IF(CAR(CDR(NSTATE)).NE.0) CALL PUSH(NSTATE,INVES)
      CALL PUSH(NSTATE,NSTATS)
C MERGE
      PREL = 0
      WOR = CAR(POIN)
```

```
      HYPO = CAR(CAR(CDR(POIN)))
      ISTRUC = CDR(NSTRUC)
      NPOIN = NPOINT (ISTRUC,WOR,HYPO)
      I = CDR(NPOIN)
      K = CDR(CDR(CAR(I)))
192   IF(CDR(I).EQ.0) GOTO 190
      IF(CAR(CDR(I)).EQ.0) GOTO 191
      I = CDR(I)
      GOTO 192
  191 CALL BACK(CDR(I))
190    CALL APPEND (I,ONEW,J)
      AANH = I
      IF(F.NE.0) GOTO 52
       J = CDR(ONEW)
53    IF(CDR(I).EQ.0) GOTO 51
      IF(CAR(CDR(I)).EQ.0) GOTO 52
      I = CDR(I)
      GOTO 53
51    CALL APPEND (I,0,I)
52    IF(COMP(ORULE,3).EQ.PREDIC) PRFL = 1
      FETS = CDR(CAR(CDR(NPOIN)))
C SYNTACTIC STATE
      CAR(NSTRUC) = NEWS
C(3) CHANGES IN HEAD CONFIGURATION
202   IF(ANEWS.NE.0) CAR( CDR(FETS)) = ANEWS
C(II) STATE IN CASE NETWORK
203   I3 = CDR(CDR(CDR(FETS)))
      IF(CASEST.NE.0) CAR(CDR(CDR(FETS))) = CASEST
C HEAD IS OBJECT
C(III) SYNTACTIC FEATURE COMPLEX
204   IF(NSYN.EQ.0) GOTO 205
      IF(CAR(I3).NE.0) CALL ERASE(CAR(I3))
      CAR(I3) = NSYN
205   IF(COMP(NRULE,2).NE.OBJEC) GOTO 206
C(IV) SEM FEATURE COMPLEX
      IF(NRES.EQ.0) GOTO 196
      IF(CAR(CDR(I3)).NE.0) CALL ERASE(CAR(CDR(I3)))
       CAR(CDR(I3)) = NRES
      GOTO 196
C HEAD IS ADJUNCT
206   IF(CHAR.NE.0) CAR(CDR(I3)) = CHAR
C VERBS
196   IF(PRFL.EQ.0) GOTO 197
      J = AANH
      I = CDR(CAR(NSTATE))
      CDR(CAR(NSTATE)) = CAR(CDR(J))
      CALL APPEND (CDR(CDR(CAR(NSTATE))),I,L)
      L = CDR(J)
      CALL BACK(L)
      CALL APPEND (J,0,J)
198   CAR(CAR(NSTATE)) = PREDIC
197   IF(TR.EQ.1) CALL PRLIST(CAR(NSTATE),8,6)
58    CONTINUE
      RETURN
C END MESSAGES
1001  IF(OU.EQ.0) RETURN
D     WRITE(6,1011)
```

```
     D1011 FORMAT (8X,'+ WRONG HEAD OR NO TRANSITION IN SYNT NET')
     D      RETURN
0320  1002  IF(OU.EQ.0) RETURN
     D      WRITE(6,1012)
     D1012 FORMAT (8X,'+ SYNTACTIC FEATURES MATCH UNSUCCESSFUL')
     D      RETURN
0322  1003  IF(OU.EQ.0) RETURN
     D      WRITE(6,1013)
     D1013  FORMAT (8X,'+ SEMANTIC FEATURES MATCH UNSUCCESFUL')
     D      RETURN
0324  1004  IF(OU.EQ.0) RETURN
     D      WRITE(6,1014)
     D1014 FORMAT (8X,'+ HEAD TAKES NO OBJECTS OR WRONG POSITION')
     D      RETURN
0326  1005   IF(OU.EQ.0) RETURN
     D      WRITE(6,1015)
     D1015 FORMAT (8X,'+MISSING CASE OR FUNCTION IN SEM NETWORK')
     D      RETURN
0328  1006  IF (OU.EQ.0) RETURN
     D      WRITE (6,1016)
     D1016 FORMAT (8X,'+NO TRANSITION IN SEM NETWORK')
     D      RETURN
0330  1007  IF (OU.EQ.0) RETURN
     D      WRITE (6,1017)
     D1017 FORMAT (8X, '+ SEMANTIC FEATURES MATCH UNSUCCESSFUL')
     D      RETURN
0332        END
```

NPOINT

parameters: STURC, WOR, HYPO

Operation:

This small auxiliary function is used to locate in a configuration (pointed at by STRUC) the information of a word (addressed by WOR) for a certain hypothesis (HYPO). The result is a pointer to a cell where the addressed configuration started.

code:

```
      INTEGER FUNCTION NPOINT (ISTRUC,WOR,HYPO)
      IMPLICIT INTEGER (A-W)
      CALL NEW(POS)
193   IF(CAR(ISTRUC).NE.WOR) GOTO 190
      IF(CAR(CAR(CDR(ISTRUC))).NE.HYPO) GOTO 190
      NPOINT = ISTRUC
1     IF(POS.EQ.0) RETURN
      CALL POPUP(I,POS)
      GOTO 1
190   ISTRUC = CDR(ISTRUC)
      IF(CDR(ISTRUC).EQ.0) GOTO 192
      IF(CAR(CDR(ISTRUC)).EQ.0)  GOTO 192
      CALL PUSH(ISTRUC,POS)
      ISTRUC = CDR(ISTRUC)
      ISTRUC = CAR(ISTRUC)
      GOTO 193
192   CALL POPUP(ISTRUC,POS)
      IF(ISTRUC.NE.0) GOTO 190
      WRITE(6,196)
196   FORMAT(1X, 'ERROR IN FINDING ATTACHPOINT IN TREE')
      CALL EXIT
      END
```

FRAMES

parameters: FEAT1, FEAT2 being two information sequences as found in a configuration
STYPE the qual/mod/undet characteristic
SEMF (optional) a semantic feature complex.

operation:

FRAMES computes whether the semantic features are compatible. Result of FRAMES is NIL if no match (neither for qual nor undet) or the resulting semantic features domain if a match was successful. Moreover FRAMES decides which characteristic holds if possible on the basis of semantic features.

code:

```
          INTEGER FUNCTION FRAMES (FEAT1,FEAT2,STYPE,SEMF)
          IMPLICIT INTEGER (A-W)
          LOGICAL*1 AF
          COMMON/CODE/ LOCK,RULE,BEFORE,AFTER,TRUE,FALSE,UNDET,FUNCTW,
     *       SYNNET,FRAME,OBJEC,UNMA,PREDIC
          COMMON/COMF/COMF(30,10)
          COMMON/COD2/MOD,QUAL,ADJU
          COMMON CAR(3000),CDR(3000),AF(3000)
          COMMON/INFTRE/SYNTRE,SEMTRE,FUNTRE
C GET CASE FRAMES
          FRAMES = 0
          IFR = 0
          IFRNAM = CAR(CDR(FEAT2))
          JFRNAM = CAR(CDR(FEAT1))
          IF (IFRNAM.EQ.0.OR.JFRNAM.EQ.0) GOTO 8
          JROLES = SEARCH (JFRNAM)
          JR = JROLES
          IROLES = SEARCH (IFRNAM)
          IF (IROLES.EQ.0.OR.JROLES.EQ.0) GOTO 8
C SEARCH FEATURES TO BE SATISFIED
          ICASE = CAR(CDR(CDR(CDR(FEAT2))))
    2     IF (CAR(CAR(IROLES)).EQ.ICASE) GOTO 3
          IROLES = CDR(IROLES)
          IF (IROLES.NE.0) GOTO 2
          GOTO 10
    3     SEMF2 = CAR(CDR(CAR(IROLES)))
    D     WRITE (6,1)
    D1    FORMAT (8X,'INVESTIGATE THE FOLLOWING SEM.FEATURES:')
    D     CALL PRLIST (SEMF2,8,6)
```

```
C SEARCH FEATURES OF SLOT FILLER
      IF (STYPE.EQ.MOD) GOTO 7
C (A) QUALIFYING
      IF(SEMF.NE.0) GOTO 6
      JCASE = CAR(CDR(CDR(CDR(FEAT1))))
4     IF (CAR(CAR(JROLES)).EQ.JCASE) GOTO 5
      JROLES = CDR(JROLES)
      IF (JROLES.NE.0) GOTO 4
      GOTO 10
5     SEMF = EXT(CAR(CDR(CAR(JROLES))))
C COMPARE
6     FRAMES = MATCH (SEMF2,SEMF,SEMTRE)
D     CALL PRLIST (SEMF,8,6)
      IF (FRAMES.EQ.0) GOTO 7
      IFR = FRAMES
C(B) MODIFYING
      IF (STYPE.EQ.QUAL) RETURN
      IF (STYPE.EQ.UNDET) STYPE = QUAL
7     ISEMF = EXT(CAR(CDR(CAR(JR))))
C COMPARE
D     CALL PRLIST (ISEMF,8,6)
      FRAMES = MATCH (SEMF2,ISEMF,SEMTRE)
      IF (FRAMES.EQ.0) GOTO 12
      IF (STYPE.EQ.QUAL) STYPE = UNDET
12    IF (IFR.NE.0) FRAMES = IFR
      RETURN
C ERRORS
8     WRITE (6,9)
9     FORMAT (1X, 'MISSING FRAME')
      RETURN
10    WRITE (6,11)
11    FORMAT (1X,'MISSING CASE IN FRAME ')
      RETURN
      END
```

3.3. The computation of the structures

We present now three subroutines which extract the linguistic informationstructures defined earlier from the particles. The implementation of this subroutines is mainly due to K. De Smedt.

(i) Functional structures

FUN

parameters: CONF (a configuration)

operation:

FUN computes the functional structure and prints it on an output device

code:

```
      SUBROUTINE FUN (CONF)
      IMPLICIT INTEGER (A-W)
      LOGICAL*1 AF
        COMMON/FIN/FIN,TR
      COMMON CAR(3000),CDR(3000),AF(3000)
      IF(CONF.EQ.0) RETURN
      CALL NEW(PDS)
      CALL NEW(FUNK)
      OUTFUN=FUNK
      INWOR=CONF
    1 INFUN=CDR(CAR(CDR(INWOR)))
      J=CDR(CDR(CAR(CDR(INWOR))))
        IF (CAR(J).EQ.0.OR.CAR(J).EQ.FIN) GOTO 3
        IF (ELEM(FIN,CAR(J)).EQ.0) GOTO 50
    3   J = CAR(CDR(J))
        IF (J.EQ.0.OR.J.EQ.FIN) GOTO 6
        IF (ELEM(FIN,J).EQ.0) GOTO 50
    6   CAR(OUTFUN)=CAR(INFUN)
      INW=CDR(CDR(INWOR))
      IF((INW.EQ.0).OR.(CAR(INW).EQ.0)) GOTO 2
      CALL NEW(OUTWOR)
      CAR(OUTWOR)=CAR(INWOR)
      CALL APPEND(OUTFUN,OUTWOR,I3)
    5 INWOR=CAR(INW)
      CALL NEW(OUTFUN)
      CALL APPEND(OUTWOR,OUTFUN,IX)
      INW=CDR(INW)
      IF((INW.EQ.0).OR.(CAR(INW).EQ.0)) GOTO 1
      CALL PUSH(IX,PDS)
      CALL PUSH(INW,PDS)
      GOTO 1
```

```
      2 CALL APPEND(OUTFUN,CAR(INWOR),OUTWOR)
        CALL POPUP(INW,PDS)
        IF(INW.EQ.0) GOTO 4
        CALL POPUP(OUTWOR,PDS)
        GOTO 5
      4 CALL PRLIST(FUNK,1,6)
        CALL PLOTLI(FUNK,1,1,1)
        RETURN
 50     CONF = 0
        RETURN
        END
```

(ii) Case structures

CAS

parameters: CONF , a configuration

operation:

CAS computes the case structure and prints it on an outputdevice.

code:

```
        SUBROUTINE CAS(CONF)
        IMPLICIT INTEGER(A-W)
        LOGICAL*1 AF
        COMMON CAR(3000),CDR(3000),AF(3000)
        COMMON /COD2/ MOD,QUAL,ADJU
        COMMON /COMF/ COMF(30,10)
        COMMON /CODE/ LOCK,RULE,BEFORE,AFTER,TRUE,FALSE,UNDET,FUNCTW,
     *  SYNNET,FRAME,OBJEC,UNMA,PREDIC
        CAST = FRAME
        IF(CONF.EQ.0) RETURN
        CALL NEW(CASE)
        CS=CASE
        CAR(CS)=CAST
        CALL PUSH(CONF,PDSP)
        CALL PUSH(0,PDST)
        FL = 0
      1 CALL POPUP(P,PDSP)
        CALL POPUP(T,PDST)
        IF(P.EQ.0) GOTO 90
        FLAG=0
        PFU = CDR(CAR(CDR(P)))
        CALL GET(CAR(PFU),RULE,IR)
        PINW=CDR(P)
        IF(FL.EQ.0) GOTO 2
        IF(COMF(IR,2).NE.OBJEC) GOTO 11
 2      IF(COMF(IR,2).EQ.OBJEC) GOTO 12
        CALL GET(CAR(P),CAR(CAR(CDR(P))),HYP)
        SUBJ = CAR(CDR(CDR(CDR(CAR(CDR(HYP))))))
```

```
     12 FL=1
      5 PINW=CDR(PINW)
        IF((PINW.EQ.0).OR.(CAR(PINW).EQ.0)) GOTO 1
        P2=CAR(PINW)
     17 P2FU=CDR(CAR(CDR(P2)))
        CALL GET(CAR(P2FU),RULE,IR)
        IF(COMF(IR,2).NE.OBJEC) GOTO 6
        P2CA=CAR(CDR(CDR(CDR(CDR(CDR(P2FU))))))
        IF(P2CA.EQ.0) P2CA = SUBJ
        IF(FLAG.EQ.1) GOTO 4
        CALL NEW(TX)
        CALL APPEND(CS,TX,CS)
        CAR(TX)=CAR(P)
      4 CALL NEW(MX)
        CALL APPEND(TX,MX,TX)
        CAR(MX)=P2CA
        CALL APPEND(MX,CAR(P2),MX)
        FLAG=1
        P2NW=CDR(CDR(P2))
        IF((P2NW.EQ.0).OR.(CAR(P2NW).EQ.0)) GOTO 18
        CALL PUSH(P2,PDSP)
        CALL PUSH(P,PDST)
     18 IF(PDSP2.NE.0) GOTO 15
        GOTO 5
6       IF(COMF(IR,2).NE.ADJU) GOTO 14
        CALL PUSH(P2,PDSP)
        CALL PUSH(P,PDST)
        IF(PDSP2.NE.0) GOTO 15
        GOTO 5
     14 IF(COMF(IR,2).NE.FUNCTW) CALL PRLIST(COMF(IR,2),0,6)
        CALL PUSH(P2NWFW,PDSP2)
        P2NWFW=CDR(P2)
     15 P2NWFW=CDR(P2NWFW)
        IF((P2NWFW.EQ.0).OR.(CAR(P2NWFW).EQ.0)) GOTO 16
        P2=CAR(P2NWFW)
        GOTO 17
     16 CALL POPUP(P2NWFW,PDSP2)
        IF(PDSP2.NE.0)GOTO 15
        GOTO 5
11      IF(COMF(IR,2).NE.ADJU) GOTO 1
        CALL GET(CAR(P),CAR(CAR(CDR(P))),HYP)
        VIEWP = CAR(CDR(CDR(CDR(CAR(CDR(HYP))))))
        IF(FLAG.EQ.1) GOTO 13
        CALL NEW(TX)
        CALL APPEND(CS,TX,CS)
        CAR(TX)=CAR(P)
     13 CALL NEW(MX)
        CALL APPEND(TX,MX,TX)
        CAR(MX)=VIEWP
          CALL APPEND (MX,CAR(T),MX)
          FLAG = 1
        GOTO 2
     90 CALL PRLIST(CASE,1,6)
D       CALL PLOTLI(CASE,1,1,1)
        RETURN
        END
```

(iii)Semantic structure

SEM

parameters:CONF, a configuration

operation:

SEM computes the semantic structure and prints it on an outputdevice

code:

```
            SUBROUTINE SEM(CONF)
            IMPLICIT INTEGER(A-X)
            LOGICAL*1 AF
            COMMON CAR(3000),CDR(3000),AF(3000)
            COMMON/SEM/OLIST,SEMSTR,PRED,ARG,FEAT,MOD,OBJEC,ADJU,FUNCTW
            COMMON/COMF/COMF(30,10)
            COMMON/ADD/RULE
D           NUM=0
            P2NWFW=0
D           WRITE(6,101)
D   101     FORMAT(1X,'CREATING TOP OF SEMANTIC STRUCTURE')
            CALL NEW(SEMA)
            CAR(SEMA)=SEMSTR
            SM=SEMA
D           WRITE(6,102)
D   102     FORMAT(1X,'CREATING INITIAL TASK IMAGE')
            CALL PUSH(CONF,PDSCO)
            CALL PUSH(0,PDSSE)
            CALL PUSH(0,PDSOX)
            CALL PUSH(0,PDSPR)
          1 CALL POPUP(PCO,PDSCO)
D           NUM=NUM+1
D           WRITE(6,146) NUM
D   146     FORMAT(1H0,1H[,I2,1H])
D           WRITE(6,103)
D   103     FORMAT(1H0,'.I. POPPING UP NEW TASK IMAGE')
D           WRITE(6,104)
D   104     FORMAT(5X,'PRESENT POINT IN CONFIGURATION:')
D           CALL PRLIST(PCO,9,6)
            CALL POPUP(PSE,PDSSE)
```

```
D       WRITE(6,105)
D 105 FORMAT(5X,'ATTACHMENT POINT IN SEMANTIC STRUCTURE:')
D       CALL PRLIST(PSE,9,6)
        CALL POPUP(MQOX,PDSOX)
D       WRITE(6,106)
D 106 FORMAT(5X,'TOP OF NODE (FOR QUAL):')
D       CALL PRLIST(MQOX,9,6)
        CALL POPUP(MQPR,PDSPR)
D       WRITE(6,107)
D 107 FORMAT(5X,'PREDICATE NODE (FOR MOD):')
D       CALL PRLIST(MQPR,9,6)
        IF(PCO.EQ.0) GOTO 90
        IF(PSE.EQ.0) GOTO 17
     18 IF(CDR(PSE).NE.0) PSE=CDR(PSE)
        IF(CDR(PSE).NE.0) GOTO 18
D       WRITE(6,109)
D 109 FORMAT(5X,'READJUSTED ATTACHMENT POINT:')
D       CALL PRLIST(PSE,9,6)
     17 PFU=CDR(CAR(CDR(PCO)))
        CALL GET(CAR(PFU),RULE,IR)
D       WRITE(6,110)
D 110 FORMAT(1H0,'.II. EXECUTION OF TASK')
D         CALL PRLIST (CAR(PFU),30,6)
D       WRITE(6,111)
D 111 FORMAT(1H+,'FUNCTION OF PRESENT WORD IS:')
        PNW=CDR(PCO)
        IF(PSE.NE.0) GOTO 19
D       WRITE(6,113)
D 113 FORMAT(1X,'* PRESENT WORD IS FIRST WORD IN CONFIGURATION'/
D     *5X,'STARTING TO CREATE INITIAL OBJECT NODE')
        IF(COMF(IR,2).EQ.OBJEC) OLIST=CDR(OLIST)
        P2=PCO
        P2FU=PFU
        GOTO 16
      2 PSE=NPL
        MQOX=OX
        MQPR=PR
D       WRITE(6,114)
D 114 FORMAT(3X,'CHANGING TASK IMAGE AFTER CREATION OF NODE')
D       WRITE(6,115)
D 115 FORMAT(5X,'ATTACHMENT POINT IN SEMANTIC STRUCTURE:')
D       CALL PRLIST(PSE,9,6)
D       WRITE(6,116)
D 116 FORMAT(5X,'TOP OF NODE (FOR QUAL):')
D       CALL PRLIST(MQOX,9,6)
D       WRITE(6,117)
D 117 FORMAT(5X,'PREDICATE NODE (FOR MOD):')
D       CALL PRLIST(MQPR,9,6)
D       WRITE(6,120)
D 120 FORMAT(1X,'STARTING TO TRACE DEPENDENT WORDS')
        GOTO 4
     19 IF(COMF(IR,2).NE.OBJEC) GOTO 12
D       CALL PRLIST(CAR(PCO),15,6)
D       WRITE(6,118)
D 118 FORMAT(1H+,'PRESENT WORD:              IS OBJECT-TYPE'/
D     *1X,'STARTING TO TRACE DEPENDENT WORDS')
      4 PNW=CDR(PNW)
        IF((PNW.EQ.0).OR.(CAR(PNW).EQ.0)) GOTO 80
        P2=CAR(PNW)
D       CALL PRLIST(CAR(P2),27,6)
D       WRITE(6,119)
D 119 FORMAT(1H+,' => DEPENDENT WORD FOUND:')
```

```
      25 P2FU=CDR(CAR(CDR(P2)))
         CALL GET(CAR(P2FU),RULE,IR)
D        CALL PRLIST(CAR(P2FU),18,6)
D        WRITE(6,121)
D   121  FORMAT(1H+,4X,'FUNCTION IS:')
         IF(COMF(IR,2).NE.OBJEC) GOTO 7
D        CALL PRLIST(CAR(P2),11,6)
D        WRITE(6,122)
D   122  FORMAT(1H+,4X,'WORD:                      IS OF OBJECT-TYPE'/
D       *5X,'STARTING TO CREATE NEW OBJECT NODE')
      16 CALL NEW(NPL)
         OX=CAR(OLIST)
         OLIST=CDR(OLIST)
         CAR(NPL)=OX
         CALL APPEND(SM,NPL,SM)
         CALL NEW(PR)
         CAR(PR)=PRED
         CALL APPEND(NPL,PR,NPL)
         CALL GET(CAR(P2),CAR(CAR(CDR(P2))),INF)
         IHPR=CDR(INF)
         CALL APPEND(PR,CAR(CDR(CDR(IHPR))),PR)
         CALL APPEND(PR,CAR(IHPR),PR)
         IF(CAR(CDR(IHPR)).NE.0) CALL APPEND(PR,CAR(CDR(IHPR)),PR)
         FEAIN=CDR(CDR(CDR(P2FU)))
         IF(COMF(IR,2).NE.OBJEC) FEAIN=CDR(FEAIN)
         CALL FEACOM(CAR(FEAIN),FEAOUT)
         IF(FEAOUT.EQ.0) GOTO 20
         CALL NEW(FE)
         CAR(FE)=FEAT
         CALL APPEND(NPL,FE,NPL)
         CALL APPEND(FE,FEAOUT,FE)
D        WRITE(6,123)
D   123  FORMAT(1X,'* OBJECT NODE COMPLETED AND ATTACHED TO ',
D       *'SEMANTIC STRUCTURE')
D        CALL PRLIST(CAR(SM),7,6)
      20 P2CA=CDR(CDR(CDR(CDR(CDR(P2FU)))))
         IF((COMF(IR,2).NE.OBJEC).OR.(CAR(P2CA).EQ.0)) GOTO 2
D        CALL PRLIST(OX,27,6)
D        WRITE(6,124)
D   124  FORMAT(1H+,4X,'NOW ATTACHING OBJECT:           TO ARGUMENTS')
         IF(CAR(CAR(PSE)).EQ.ARG) GOTO 5
         CALL NEW(AR)
         CAR(AR)=ARG
         CALL APPEND(PSE,AR,PSE)
       5 CALL NEW(CA)
         CAR(CA)=CAR(P2CA)
         CALL APPEND(AR,CA,AR)
         CALL APPEND(CA,OX,CA)
D        CALL PRLIST(CAR(PSE),7,6)
         P2NW=CDR(CDR(P2))
D        CALL PRLIST(CAR(P2),5,6)
         IF((P2NW.EQ.0).OR.(CAR(P2NW).EQ.0)) GOTO 29
D        WRITE(6,125)
D   125  FORMAT(1H+,17X,'HAS DEPENDENT WORDS - PUSH NEW TASK IMAGE')
         CALL PUSH(NPL,PDSSE)
         CALL PUSH(P2,PDSCO)
         CALL PUSH(OX,PDSOX)
         CALL PUSH(PR,PDSPR)
         GOTO 27
```

```
         29 CONTINUE
D           WRITE(6,126)
D       126 FORMAT(1H+,17X,'HAS NO DEPENDENT WORDS')
         27 IF(PDSP2.NE.0) GOTO 24
            GOTO 4
7             CALL GET (CAR(P2FU),RULE,IR)
              IF (COMF(IR,2).NE.ADJU) GOTO 8
            CHAR=CAR(CDR(CDR(CDR(CDR(P2FU)))))
D           CALL PRLIST(CAR(P2),11,6)
D           WRITE(6,128)
D       128 FORMAT(1H+,4X,'WORD:                    IS OF ADJUNCT-TYPE')
D           CALL PRLIST(CHAR,16,6)
D           WRITE(6,129)
D       129 FORMAT(1H+,6X,'SUBTYPE:             - PUSHING NEW TASK IMAGE')
            IF(CHAR.EQ.MOD) GOTO 21
            CALL PUSH(PSE,PDSSE)
            GOTO 6
         21 CALL PUSH(MOPR,PDSSE)
          6 CALL PUSH(P2,PDSCO)
            CALL PUSH(MOOX,PDSOX)
            CALL PUSH(0,PDSPR)
            IF(PDSP2.NE.0) GOTO 24
            GOTO 4
          8 IF(COMF(IR,2).NE.FUNCTW) GOTO 23
D           CALL PRLIST(CAR(P2),11,6)
D           WRITE(6,131)
D       131 FORMAT(1H+,4X,'WORD:                    IS OF FUNCTIONWORD-TYPE')
            CALL PUSH(P2NWFW,PDSP2)
              P2NWFW = CDR(P2)
         24 P2NWFW=CDR(P2NWFW)
            IF((P2NWFW.EQ.0).OR.(CAR(P2NWFW).EQ.0)) GOTO 26
            P2=CAR(P2NWFW)
D           CALL PRLIST(CAR(P2),13,6)
D           WRITE(6,132)
D       132 FORMAT(1H+,6X,'WORD:                    IS DEPENDENT FROM FUNCTIONWORD'/
D          *13X,'AND IS CONSIDERED TO TAKE ITS PLACE')
            GOTO 25
         26 CALL POPUP(P2NWFW,PDSP2)
              IF (PDSP2.NE.0) GOTO 24
D           WRITE(6,133)
D       133 FORMAT(7X,'- NO (MORE) WORDS DEPENDENT FROM FUNCTIONWORD')
D           WRITE(6,134)
D       134 FORMAT(1H+,53X,'- PDS EMPTY')
            GOTO 4
         23 CALL PRLIST(CAR(P2FU),39,6)
            WRITE(6,135)
        135 FORMAT(1H+,'$ ERROR $ --CANNOT IDENTIFY FUNCTION:')
            CALL PRLIST(CAR(P2),39,6)
            WRITE(6,136)
        136 FORMAT(1H+,29X,'OF WORD:')
            GOTO 4
         12 IF(COMF(IR,2).NE.ADJU) GOTO 13
D           CALL PRLIST(CAR(PCO),15,6)
D           WRITE(6,137)
D       137 FORMAT(1H+,'PRESENT WORD:                   IS OF ADJUNCT-TYPE')
            OX=CAR(CDR(CDR(CDR(CDR(PFU)))))
D           CALL PRLIST(OX,14,6)
D           WRITE(6,138)
D       138 FORMAT(1H+,2X,'SUBTYPE:'/
D          *5X,'STARTING TO CREATE NEW ADJUNCT NODE')
```

```
              CALL NEW(NPL)
              CAR(NPL)=OX
              CALL APPEND(PSE,NPL,PSE)
              CALL NEW(PR)
              CAR(PR)=PRED
              CALL APPEND(NPL,PR,NPL)
                CALL GET (CAR(PCO),CAR(CAR(CDR(PCO))),INF)
                IHPR = CDR(INF)
              CALL APPEND(PR,CAR(CDR(CDR(IHPR))),PR)
              CALL APPEND(PR,CAR(IHPR),PR)

              IF(CAR(CDR(IHPR)).NE.0) CALL APPEND(PR,CAR(CDR(IHPR)),PR)
              IF(OX.EQ.MOD) GOTO 2
D             CALL PRLIST(MQOX,20,6)
D             WRITE(6,139)
D      139 FORMAT(1H+,'NOW ATTACHING TOP:         TO ARGUMENTS OF QUALIFIER')
              CALL NEW(AR)
              CAR(AR)=ARG
              CALL APPEND(NPL,AR,NPL)
              CALL NEW(CA)
              CAR(CA)=CAR(CDR(CDR(IHPR)))
              CALL APPEND(AR,CA,AR)
              CALL APPEND(CA,MQOX,CA)
D             CALL PRLIST(CAR(NPL),1,6)
D             CALL PRLIST(OX,3,6)
D             WRITE(6,141)
D      141 FORMAT(1H+,1H*,7X,'NODE COMPLETED AND ATTACHED')
D             CALL PRLIST(CAR(PSE),1,6)
              GOTO 2
         13 CALL PRLIST(CAR(PFU),39,6)
D             WRITE(6,142)
D      142 FORMAT(1H+,'$ ERROR $ --CANNOT IDENTIFY FUNCTION:')
D               CALL PRLIST (CAR(PCO),39,6)
D               WRITE (6,143)
D      143 FORMAT(1H+,29X,'OF WORD:'/
D            *12X,'OR INCORRECT INPUT FROM POPUP')
              GOTO 4
         80 CONTINUE
D             WRITE(6,144)
D      144 FORMAT(1X,'- NO (MORE) WORDS DEPENDENT FROM PRESENT WORD'/
D            *1H0,'.III. SEMANTIC STRUCTURE AT PRESENT STAGE:'/)
D             SMA=SEMA
D        81 SMA=CDR(SMA)
D             IF(SMA.EQ.0) GOTO 82
D             CALL PRLIST(CAR(SMA),7,6)
D             GOTO 81
         82 GOTO 1
         90 WRITE(6,145)
        145 FORMAT(1H0,'>>>>> SEMANTIC STRUCTURE COMPLETED NOW'/
            *7X,'FINAL OUTPUT:'/)
              SMA=SEMA
         91 SMA=CDR(SMA)
              IF(SMA.EQ.0) RETURN
              CALL PRLIST(CAR(SMA),7,6)
              CALL PLOTLI(CAR(SMA),1,1,1)
              GOTO 91
              END
```