ASPECTS OF A MODULAR THEORY OF LANGUAGE

VOL II.

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 to the parser
2.1.1. Particles
2.1.2. The parsing predicates and their combination
2.1.3. The creation of new particles
2.1.4. The general control structure
2.1.5. Example
2.1.6. The computation of the resulting structures

2.2. The production process

2.2.0. Introduction
2.2.1. The tasks
2.2.2. The process
2.2.3. Example
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 inductive 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 implications 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.
introduction

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 conditions' (ibid. 2. 18) and a right part with 'forms or schemata' upon which the evaluation can take place.

- 2.3. -
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 transformational 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 parallelism 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-Podar 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 semantics-directed parsing does not use the intermediary step of having preliminary structures in which functional relations or category information plays a role. Here one starts immediately 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 all available knowledge directed parsing. 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:
introduction

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

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):

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.

There are some general conditions for the combination of two particles, such as (i) particles created due to the same input pulse are never combined (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 around in the state space. As a whole you get a regular pulse of incoming energy creating particles, and of subsequent combination processes.
Now comes the second part of the story. Imagine a second work space which we will call the cognitive 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:

\[ \text{inputpulse for cognitive space} \]

\[ \text{INPUT for state space.} \]
Introduction

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 analysis actions. The second type (where particles themselves become forces) the interpretation actions. 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 ambiguity 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:

\[
\langle a_1, \ldots, a_{n+2} \rangle \quad n \geq 0
\]

such that

\( a_1 \) is a word
\( a_2 \) is an information sequence
\( a_{i+2}, \ldots, a_{n+2} \) for \( i \geq 0, n \geq 1 \) other configurations

**Definition**

An information sequence \( i \) for adjuncts and function words is a 6-tuple:

\[
i = \langle i_1, i_2, i_3, i_4, i_5, i_6 \rangle
\]

such that

\( i_1 \) is the hypothesis of the word under consideration; we number hypotheses according to the moment of input: \( \text{INF1, INF2, ...} \)
\( i_2 \) is the function name of the word for that hypothesis
\( i_3 \) the state in syntactic network according to our principle of the preservation of ambiguity we allow there to be a set of states;
\( i_4 \) the state in the semantic network, also here we will allow there to be a set of states;
\( i_5 \) the internal syntactic feature complex (the extension)
\( i_6 \) the qual/mod/undet characteristic

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

\[
i = \langle i_1, i_2, i_3, i_4, i_5, i_6, i_7 \rangle
\]

such that

\( i_1, i_2, i_3, i_4, i_5 \) are as for adjuncts
\( i_6 \) is the extension of the semantic features associated with the viewpoint of the word for the predicate in the lexicon sequence that immediately caused this information sequence
\( i_7 \) 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 incorporated 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.

**Example**

1. ((NL) LETTER (INP4 NOM.OBJ NIL NIL ((SING.OBJ)(SING.3PS)))
   state word hypo function state state
   in in synt. state
   ((THING) NIL) synt. synt.
   semantic case 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 synt.
   state in sem.
   syntactic features characteristic
   netw.

   (configuration for adjunct)

3. ((NS) 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:

- 2.15 -
Particles

Convention

If \( c = (a_1, a_2, a_3, ..., a_{n+2}) \) is a configuration with 
\( a_3, ..., a_{n+2} \) other configurations then we draw a tree:

![Tree Diagram]

We can now define the particles themselves:

Definition

A particle is a quadruple \( (a_1, a_2, a_3, a_4) \) with

- \( a_1 \) the range (i.e. from where to where in the input sequence the particle contains words)
- \( a_2 \) LOCKED or NIL (keywords indicating whether the particle is no longer or still subject to combination processes)
- \( a_3 \) a state in a network or a set of states associated with the word in the topconfiguration of \( a_4 \)
- \( a_4 \) a configuration.

Convention

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 threshold 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 threshold and weight as follows:
Definition

"Let $\Phi = \phi_1, \phi_2, \ldots, \phi_n$ be a family of predicates.

We will say that $\psi$ is linear with respect to $\Phi$ if there exists a number $\theta$ (the threshold) and a set of numbers $a_{\phi_1}, a_{\phi_2}, \ldots, a_{\phi_n}$ (the weights) such that

$$\psi(X) = \begin{cases} 1 & \text{if } a_{\phi_1} \phi_1(X) + \ldots + a_{\phi_n} \phi_n(X) \geq \theta \end{cases}$$  

(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 $\Phi$ 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 FARSING 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 threshold 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 wᵢ as iᵢₖ and the n-th item in it as iₙ,ₖ.

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 \( w_1 \) (in configuration \( c_1 \)) and \( w_2 \) (in configuration \( c_2 \)), if \( w_1 \) is supposed to have a particular grammatical function \( f \) as regards \( w_2 \), \( w_2 \) should have a particular possible function, indicated by \( \text{function-of-head} (f) \).

From this we extract the following predicate:

**Definition**

\[
p\text{-function-of-head}: W \times W \rightarrow \{ \text{TRUE, FALSE} \}
\]

is defined for \((Vw) (i_2, w_1 \in F\text{-adj} \cup F\text{-func}t)\) as follows:

\[
p\text{-function-of-head}(w_1, w_2) = \begin{cases} 
\text{TRUE} & \text{if } \text{function-of-head} (i_2, w_1) = i_2, w_2 \\
\text{FALSE} & \text{otherwise}
\end{cases}
\]

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**

\[
p\text{-taking-objects}: W \times W \rightarrow \{ \text{TRUE, FALSE} \}
\]

is defined for \((Vw1) (i_2, w_1 \in F\text{-object})\) as follows:

\[
p\text{-taking-objects}(w_1, w_2) = \begin{cases} 
\text{TRUE} & \text{if } \text{taking-objects}(i_2, w_2) = \text{TRUE} \\
\text{FALSE} & \text{otherwise}
\end{cases}
\]

- 2.22. -
(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 \( w_i \prec w_j \) if in the input sequence we have \( w_1 \ldots w_i \ldots w_j \ldots w_n \) \( n > 0 \) and \( 1 \leq i < j \leq n \).

Definition

Let \( p\text{-position} : W \times W \to \{ \text{TRUE, FALSE} \} \) be defined for \( (\forall w_l) \ (l_2, w_l \in \text{F-adjuncts } \cup \text{F-functw} \) as follows:

\[
\begin{align*}
p\text{-position} (w_1, w_2) &= \begin{cases} 
\text{TRUE} & \text{if } \text{position}(l_2, w_1) = \text{before or undet and } w_1 \prec w_2 \\
\text{FALSE} & \text{otherwise}
\end{cases}
\end{align*}
\]

Definition

Let \( p\text{-object-position} : W \times W \to \{ \text{TRUE, FALSE} \} \) be defined for \( (\forall w_l) \ (l_2, w_l \in \text{F-object}) \) as follows:

\[
\begin{align*}
p\text{-object-position} (w_1, w_2) &= \begin{cases} 
\text{TRUE} & \text{if } \text{object-position}(l_1, w_2) = \text{before or undet and } w_1 \prec w_2 \\
\text{FALSE} & \text{otherwise}
\end{cases}
\end{align*}
\]
(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

\[ p\text{-synt-network}: W \times W \rightarrow \{\text{TRUE, FALSE}\} \] is defined

\[
(V \ w2) \text{(syntactic-network } (i_2, w2) \text{ is defined) as follows: Let } S = s_1, \ldots, s_n \text{ be the set of states associated with the particle of } w_2, \text{ then }
\]

\[
p\text{-synt-netw} (w1, w2) = \begin{cases} 
\text{TRUE if } (3 s \in S) (\gamma (i_2, w1, s) \neq \emptyset) \\
\text{FALSE otherwise}
\end{cases}
\]

- 224. -
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

\[ p\text{-concord}: W \times W \rightarrow \{ \text{TRUE, FALSE} \} \]

is a function defined

\[ (\forall w_1) (w_1 \in P\text{-object}) \]

\[ p\text{-concord}(w_1,w_2) = \begin{cases} 
\text{TRUE if either} \\
(1) \text{concord}(i_2,w_1) = \text{false} \\
\text{or} \\
\text{concord}(i_2,w_1) = \text{true and} \\
\text{syntactic-feature-complex of } w_2 \\
\text{matches with } i_2,w_1 \\
\text{FALSE otherwise} 
\end{cases} \]

(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 distinction), (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.

Definition

\[ p-sem.feat-adju : W \times W \rightarrow \{ \text{TRUE}, \text{FALSE} \} \]

is defined as follows:

Let \( w_1, w_2 \in F \)-adjuncts then

\[ p-sem.feat-adju (w_1, w_2) = \begin{cases} 
\text{TRUE} & \text{if } \\
& \text{(i) either } F \text{ has the modifier/undet characteristic} \\
& \text{and } \text{match}(\text{valuerestriction}(\text{self}, p_2), \\
& \text{valuerestriction}(c_1, p_1)) = \text{TRUE} \\
\end{cases} \]

or

\[ \text{FALSE otherwise} \]

\[ \text{TRUE} \]

\[ \text{TRUE} \]
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**

\[\text{p-sem-netw} : W \times W \rightarrow \{\text{TRUE}, \text{FALSE}\}\]

is defined \((\forall w_1) (w_1 \in F\text{-objects})\) as follows:

Let \(S = \{s_1, \ldots, s_n\}\) be the set of states in the case networks with the configuration of \(w_2\), then

\[
P\text{-sem-netw} (w_1, w_2) = \begin{cases} 
\text{TRUE} & \text{if } (\exists s \in S) (\chi(i, w_1, s) = \emptyset) \\
\text{FALSE} & \text{otherwise} 
\end{cases}
\]

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.
parsing predicates

(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

\[ \text{p-sem.feat-obj: } W \times W \rightarrow \{ \text{TRUE, FALSE} \} \]

is defined \((\forall w1)\) \((w1 \in \text{P-object})\) as follows:

Let \((w1, w2) \in f, p1 = \text{predicate} (w1), c1 = \text{viewpoint} (w1), p2 = \text{predicate} (w2), \) and \(c\) one of the cases of \(p2\), then

\[
\text{p-sem.feat-obj} \ (w1, w2) = \begin{cases} 
\text{TRUE} & \text{if} \\
& \text{match} \ (\text{valuerestriction}(c, p2), \\
& \text{valuerestriction}(c1, p1) ) = \text{true}
\end{cases}
\]

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 language

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 parsing process 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 "w1 w2 w3 w4 w5" then if there are e.g. particles on w3 and w5 containing the structures

```
   w3
  /  \\
w2   w1
   \  /  and  \\
      \   \\
        w4  w5
```

(particle 1)  (particle 2)
then we may consider the merging of these two which may lead to

\[
\begin{array}{c}
\text{w5} \\
\text{w3} \\
\text{w2} \\
\text{w1}
\end{array}
\quad \text{or} \quad
\begin{array}{c}
\text{w3} \\
\text{w2} \\
\text{w1} \\
\text{w5} \\
\text{w4}
\end{array}
\]

But suppose we have particles on w2 and w5 with structures

\[
\begin{array}{c}
\text{w2} \\
\text{w1}
\end{array}
\quad \text{and} \quad
\begin{array}{c}
\text{w5} \\
\text{w4}
\end{array}
\]

\[(\text{particle 1}) \quad \text{and} \quad (\text{particle 2})\]

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

\[
\text{boy} \\
\text{the} \quad \text{good} \quad \text{old}
\]

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

\[
\begin{array}{c}
\text{boy} \\
\text{the} \quad \text{the} \quad \text{good} \quad \text{the} \quad \text{old}
\end{array}
\]

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 !)

- 2.30. -
(2) The time dimension

Another consequence of taking this time dimension seriously is that if a particle will be attached to one of the sub-configurations 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:

**Principle 2**

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

```
  book
 /    \
a     nice
```

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

\[
\begin{align*}
  a_i &= \langle a_1, i, a_2, i, a_2+1, i, \ldots, a_2+m, i \rangle & m &> 0 \\
  a_j &= \langle a_1, j, a_2, j, a_2+1, j, \ldots, a_2+n, j \rangle & n &> 0
\end{align*}
\]

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.
Definition

We say that two configurations \( a_j, a_l \) merge iff either 
\[ d-merge(a_j, a_l) \] 
or \( a_j \) merges with \( a_2+p, 1 \leq p \leq m \). The resulting configuration is denoted as \( \text{merge} (a_j, a_l) \).

Example

Given

\[
\begin{align*}
 & a_l \\
 \triangleleft & a_2+1, \ldots a_2+m, i \\
 & a_{2+1,j} \ldots a_{2+n,j}
\end{align*}
\]

and

\[
\begin{align*}
 & a_j \\
 \triangleleft & a_2+1, \ldots a_2+m, i \\
 & a_{2+1,j} \ldots a_{2+n,j}
\end{align*}
\]

then

\[
\begin{align*}
 & a_l \\
 \triangleleft & a_2+1, i, \ldots a_2+m, i \\
 \triangleleft & a_{2+1,j} \ldots a_{2+n,j}
\end{align*}
\]

\( \in \text{merge} (a_j, a_l) \)

Now we can define the merging of two particles

Definition

Let \( p_1 = \langle P_{1,1}, P_{2,1}, P_{3,1}, P_{4,1} \rangle \) and

\[
p_2 = \langle P_{1,2}, P_{2,2}, P_{3,2}, P_{4,2} \rangle
\]

be two particles then

\( p_3 \in \text{merge} (p_2, p_1) \) if

\[
p_3 = \langle P_{1,1}^tP_1, P_{2,3}, P_{3,3}, P_{4,3} \rangle \quad \text{(for } p_2, p_3 \text{ and } p_3, p_3 \text{ cf.infra)}
\]

and \( p_4, 3 \in \text{merge} (p_4, p_4, 1) \).
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 input pulse 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 word. If there are no input words 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.)
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

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

type: functionword
function-of-head and position are specified via the syntactic networks associated with nom.obj.
send-through: true.

(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 more interesting)

2. The syntactic networks

There is one left-going network and one right-going network:

for nom.obj:

\[ \text{OBJ/1} \xrightarrow{\text{det}} \text{OBJ/2} \xrightarrow{\text{casesi}} \text{OBJ/3} \]

where OBJ/1 is the initial state.

and

\[ \text{A/1} \xrightarrow{\text{nom.obj}} \text{FIN} \]

for nom.adv.adj and nom.att.adj. FIN is the final state.
(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:
-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
(ii) ARROW

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

```
XOR
AND
SING
```

(iii) FLIES

-a-

function: nom.obj
predicate: insect
subpredicate: flying
viewpoint: self
syntactic feature complex:

```
XOR
AND
PLURAL 3PS
```

-b-

function: verb
predicate: move
subpredicate: through-air
viewpoint: agent
syntactic feature complex:

```
NOT
AND
OBJECTIVE SING
```

internal feature complex: PRESENT
We now start a discussion of the parsing process. 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.

I. Initial particles

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

(i) Particle 1 (for function nom.obj) has configuration

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

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) Particle 2 (for function verb) has configuration:

\[
\text{TIME (INP2 VERB NIL NIL ((PRESENT)) UNDET )}
\]

II. Merging

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

\[
\begin{array}{c}
\text{O}_1 \\
\text{O}_2
\end{array}
\]

- 2.44 -
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.

   - 2.45. -
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 semantic 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)

(l) 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

\[
\begin{array}{c}
\text{INS/1} \\
\text{[kind]} \\
\text{FIN}
\end{array}
\]

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 (particle 6) can now be created:

\[
(\text{FLIES} \ (\text{INP3} \ \text{NOM.OBJ} \ nil \ fin \ ((\text{OBJECTIVE} \ \text{PLURAL} \ 3PS))) \\
(\text{PLURAL} \ 3PS)) \ ((\text{ANIMATE} \ )) \ nil) \\
(\text{TIME} \ (\text{INP1} \ \text{NOM.OBJ} \ fin \ nil \ ((\text{OBJECTIVE} \ \text{SING} \ 3PS))) \\
((\text{PROPERTY})) \ \text{KIND})
\]

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 test, 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'.

- 2.47 -
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:

**particle 7**

(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
  1  3
  2  4

  5  7

to
  1  2
  3  4
  5  7
```

- 2.48. -
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:

\[ \begin{array}{c}
5 \\
\circ \\
6 \\
\circ \\
7
\end{array} \]

**I. Creation of new particles**

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

**particle 8** with configuration:

\[
\text{LIKE (INP5 CASESI NIL NIL NIL )}
\]

**particle 9** with configuration:

\[
\text{LIKE (INP6 NOM.ATT.ADJ A/1 NIL UNDET )}
\]

**particle 10** with configuration:

\[
\text{LIKE (INP7 NOM.ADV.ADJ A/1 NIL (PRESENT) UNDET )}
\]

**particle 11** with configuration

\[
\text{LIKE (INP8 PREDIC NIL NIL (PRESENT) UNDET )}
\]

(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 (INF4 VERB NIL NIL ((PRESENT)) UNDET)
  (TIME (INPl NOM.OBJ NIL NIL ((SING 3PS)) ((THING)) NIL))
  (LIKE (INF7 NOM.ADV.HEAD 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 input pulse 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:

- 2.52 -
- 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 ))))

- 2.53. -
(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:

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

There is only one: particle 17 with configuration

\[(\text{AN (INP9 DET FIN)})\]

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:

```
  13    12
  |    |
  16
  |
  14
  |
  15
  |
  17
```

I. New particles

There is again only one particle: particle 18.

\[(\text{ARROW (INP10 NOM.OBJ NIL NIL})\]

\[((\text{OBJECTIVE SING 3PS})(\text{SING 3PS})) ((\text{THING})) \text{ NIL})\]
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:

particle 19 with configuration

(AARROW (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 around on the border of the range of this particle. The whole process is set in motion again by placing particle 19 on the tasklist 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

\[(\text{ARROW} \ (\text{INP}10 \ \text{NOM.OBJ} \ \text{NIL} \ \text{NIL} \ ((3\text{PS} \ \text{SING} \ \text{OBJECTIVE} \ \text{UNDEF} \ \text{LIKE})))
\ ((\text{THING})) \ \text{NIL})
\ (\text{AN} \ (\text{INP}9 \ \text{DET} \ \text{NIL}) \ (\text{LIKE}(\text{INP}5 \ \text{CASE}1 \ \text{NIL}))) \ )\]

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 FTN 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:

\[(\text{FLIES} \ (\text{INP}3 \ \text{NOM.OBJ} \ \text{NIL} \ \text{NIL} \ ((\text{OBJECTIVE} \ \text{PLURAL} \ 3\text{PS}))
\ ((\text{THING}) \ \text{NIL}))
\ (\text{TIME} \ (\text{INP}1 \ \text{NOM.OBJ} \ \text{NIL} \ \text{NIL} \ ((\text{OBJECTIVE} \ \text{SING} \ 3\text{PS})) \ ((\text{PROPERTY})) \ \text{KIND})
\ (\text{LIKE} \ (\text{INP}6 \ \text{NOM.ATT.ADJ} \ \text{FIN} \ \text{NIL} \ \text{UNDET})
\ (\text{ARROW} \ (\text{INP}10 \ \text{NOM.OBJ} \ \text{NIL} \ \text{NIL} \ ((3\text{PS} \ \text{SING} \ \text{OBJECTIVE} \ \text{UNDEF})))
\ ((\text{THING}) \ \text{TO}) \ (\text{AN} \ (\text{INP}9 \ \text{DET} \ \text{NIL}))))\]

- 2.57. -
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.

- 2.58. -
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 semantic 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 order test 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

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

(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

\[\text{- 260. -}\]
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:

Definition

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

be a configuration with

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

then

$(i_{2,k} a_{1,k})$ for $j = 0$

$F$-struct($a_k$) =

$(i_{2,k} (a_{1,k} \text{ F-struct (} a_{2+1,k}) \ldots \text{ F-struct(} a_{2+j,k}))))$ for $j > 0$

Notice that this yields a list structure which is converted 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 \( a_k = \langle a_{1,k}, a_{2,k}, a_{2+1,k}, \ldots, a_{2+j,k} \rangle \quad j \geq 0 \)

be a configuration with

\[ a_{2,k} = \langle i_{1,k}, i_{2,k}, \ldots \rangle \]

an information sequence

then

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

with

\[
\text{label} \left( \langle a_{1}, a_{2+i,k}, a_{1,k} \rangle \right) = i_{7,a_{2+i,k}}
\]

iff \( i_{2,a_{2+i,k}} \in \text{F-obj} \) for \( 1 \leq i \leq j \)

and

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

with

\[
\text{label} \left( \langle a_{1,k}, a_{1,a_{2+i,k}} \rangle \right) = i_{6,k}
\]

iff \( i_{2,k} \in \text{F-adj} \) for \( 1 \leq i \leq j \)
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

```
  NOM.OBJ
   |
   FLIES
   |
   NOM.OBJ
   |
   TIME
   |
   NOM.OBJ
   |
   ARROW
   |
   DET
   |
   AN
```
For particle 22 with configuration:

(\time (\infl2 \verb \nil nil \nil \undet ((\present) ) \undet )
  (\flies (\infl3 \nom\objc \nil \nil ((\objective \plural 3ps ))
    ((\animate) ) \what)
  (\like (\infl6 \nom\att\adj \nil \nil \undet )
    (\arrow (\infl10 \nom\objc \fin \nil ((3ps \objective \sing \undet))
      ((\thing)) \to)
    (\an (\infl9 \det \nil)))
))

functional structure:

```
VERB
  | TIME
  | NOM.OBJ
      | FLIES
      | NOM.ATT.ADJ
          | LIKE
          | NOM.OBJ
      | ARROW
      | DET
      | AN
  - 2.65. -
```
(iii) Semantic structures

The extraction of the semantic structures in the format of the SRL language is a straightforward 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 category) 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 intention 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 constitute the 'stimuli' for the production system to become active.
Definition

A **taskbuilder task** is a 4-tuple \( \langle a_1, a_2, a_3, a_4 \rangle \)
with
- \( a_1 \) = the keyword TKB (taskbuilding)
- \( a_2 \) = a pointer to the task which was the immediate source for this task
- \( a_3 \) = a pointer to a node in the semantic structure
- \( a_4 \) = 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 \( \langle a_1, a_2, a_3, a_4, a_5, a_6 \rangle \)
with
- \( a_1 \) = the keyword LEX
- \( a_2 \) = the function of the word
- \( a_3 \) = the predicate
- \( a_4 \) = the subpredicate
- \( a_5 \) = the viewpoint
- \( a_6 \) = 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 straightforward 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

-1- 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 could also be computable on beforehand.

-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 access 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 tasks 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:

First we make the initialization task pointing to the 01 node itself:

1. \( \langle \text{TKB}, \emptyset, 01, \text{NIL} \rangle \)

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 items 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:
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 \text{adv.adj}; the predicate \text{PROP} and the viewpoint \text{OF*WHAT}

Resulting tasklist:

3. \langle \text{LEX}, \text{ADV.ADJ}, \text{PROP}, \text{VERY}, \text{OF*WHAT} \rangle
2. \langle \text{LEX}, \text{ATT.ADJ}, \text{PROP}, \text{URGENT}, \text{OF*WHAT} \rangle
1. \langle \text{LEX}, \text{NOM.OBJ}, \text{WRIT}, \text{NIL}, \text{RESULT}, ((\text{SING})) \rangle

STEP 6

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

- 2.77. -
§ 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

3.2. The implementation of the parser
   3.2.1. Auxiliary routines
   3.2.2. The parser

3.3. The computation of the structures
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 separately.

Example:

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

\[
\begin{array}{c}
\text{AF} \\
\downarrow \\
\text{CAR} \\
\downarrow \\
\text{CDR}
\end{array}
\]

then the FORTRAN representation will be

<table>
<thead>
<tr>
<th></th>
<th>AF</th>
<th>CAR</th>
<th>CDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\emptyset$</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>$\emptyset$</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>$\emptyset$</td>
<td>4</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>4</td>
<td>$\emptyset$</td>
<td>C</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Note that the representation of NIL (the null list) is $\emptyset$. 

- 3.2. -
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 0. 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

<table>
<thead>
<tr>
<th></th>
<th>AF</th>
<th>CAR</th>
<th>CDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ø</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>ø</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>ø</td>
<td>4</td>
<td>ø</td>
</tr>
<tr>
<td>4</td>
<td>ø</td>
<td>7</td>
<td>ø</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>ø</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>ø</td>
<td>1ø</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>ø</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>ø</td>
<td>9</td>
<td>ø</td>
</tr>
<tr>
<td>9</td>
<td>A</td>
<td>ø</td>
<td>ø</td>
</tr>
<tr>
<td>10</td>
<td>ø</td>
<td>11</td>
<td>ø</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>ø</td>
<td>ø</td>
</tr>
<tr>
<td>12</td>
<td>ø</td>
<td>13</td>
<td>ø</td>
</tr>
<tr>
<td>13</td>
<td>C</td>
<td>ø</td>
<td>ø</td>
</tr>
</tbody>
</table>

= base register of A
= base register of B
= base register of C

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

Note that as a consequence of these options all pointers either
to lists or to atoms are of INTEGER 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.
list processing

The pointer to the freelist is called IFREE and available in a common zone 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(3000),CDR(3000),AF(3000)
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:

<table>
<thead>
<tr>
<th>CAR</th>
<th>CDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A</td>
<td>ø   = I1</td>
</tr>
<tr>
<td>2. ø</td>
<td>3   = beginning of freelist</td>
</tr>
<tr>
<td>3. ø</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

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:
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) = 0.
This yields:

<table>
<thead>
<tr>
<th>CAR</th>
<th>CDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
</tr>
<tr>
<td>Ø</td>
<td>5</td>
</tr>
<tr>
<td>Ø</td>
<td>6</td>
</tr>
</tbody>
</table>

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.
list processing

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:

```
0001 SUBROUTINE ADD (I2, I1)
0002 IMPLICIT INTEGER (A-W)
0003 LOGICAL AF
0004 COMMON CAR(3000), CDR(3000), AF(3000)
0005 NIL = 0
0006 IF(I2.EQ.,0) RETURN
0007 FLAG = 0
0008 IF(AF(I2).NE.,1) GOTO 1
0009 L = I2
0010 5
0011 J = I1
0012 IF(CAR(J).EQ.,L) GOTO 4
0013 IF(CDR(J).EQ.,0) GOTO 3
0014 CALL NEW(I)
0015 3 CALL NEW(I)
0016 CDR(J) = I
0017 CAR(I) = L
0018 GOTO 4
0019 1 FLAG = 1
0020 K = I2
0021 L = CAR(K)
0022 GOTO 5
0023 4 IF(FLAG.EQ.,0) RETURN
0024 IF(CDR(K).EQ.,0) RETURN
0025 K = CDR(K)
0026 L = CAR(K)
0027 GOTO 5
0028 END
```
list processing

ATOM

parameters: Il an atom or a list

operation: ATOM checks whether Il 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:

```
0001    LOGICAL FUNCTION ATOM (Il)
0002    IMPLICIT INTEGER (A-M)
0003    LOGICAL ! AP
0004    COMMON A,B,C,D,E,F,G,H
0005    ATOM = .FALSE.
0006    IF (AP .朱F .FALSE.) ATOM = .TRUE.
0008    END
```

- 3.9. -
list processing

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 = 

then after APPEND (I1, I2, I3)

with I3 = 2

code:

0001  SUBROUTINE APPEND(I1,I2,I3)
0002      IMPLICIT INTEGER (4,4).
0003      LOGICAL*1 AF
0004      COMMON CAR(3000), CDR(3000), AF(3000)
0005      COMMON EPREH/ EPPRE
0006      CDR(I1) = IPRE
0007      CDR(EPREH) = I2
0008      IF (EPREH.EQ.31) GOTO 1
0009      I4 = IPRE
0010      EPREH = CDR(EPPRE)
0011      CDR(I4) = 
0012      RETURN;
0014  1  WRITE(*,?)
0015  2  FLOW OUT (I1, "STORAGE EXHAUSTED IN APPEND")
0016  3  CALL EX
0017     END
list processing

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:

```
0001   SUBROUTINE ATTACH (I2, I1)
0002   IMPLICIT INTERFER (A-N)
0003   COMMON CAR(KAM), CAR(TURN), AF(3200)
0005   NIL = 0
0006   IF(I2,ED,7) RETURN
0008   J = I1
0009   C GOTO END OF LIST
0012   J = CAR(J)
0013   GOTO 2
0014   K = I2
0015   C ATTACH LIST
0018   3 IF(K=0,NIL) GOTO 4
0019   IF(CAR(K),ED,2) GOTO 4
0020   CALL NEW1(I)
0021   CAR(J) = L
0022   J = L
0023   CAR(J) = CAR(K)
0024   K = CAR(K)
0025   GOTO 3
0026   4 CAR(J) = NIL
0027   RETURN
0028   C ATTACH ATOM
0028   3 CALL NEW1(K)
0029   CAR(J) = K
0030   CAR(K) = L
0031   RETURN
0032   END
```

- 3.11. -
list processing

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:

```plaintext
0001      SUBROUTINE BACK(I)
0002      IMPLICIT INTEGER (A-Z)
0003      LOGICAL*1 I
0004      COMMON CAR(100),CH(300),AF(300)
0005      COMMON FREE*1 FREE

C THE SUBROUTINE BACK RETURNS ONE CELL TO THE FREELIST
0006      IF(I.EQ.0) GOTO 1?
0007      CAR(I) = FREE
0008      FREE(I) = N
0009      I = I + 1
0010      RETURN
0011      WRITE(*,11)
0012      11 FORMAT (1x, 'NIL IN BACK')
0013      END
```
list processing

COPY

parameters: I, a list

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

code:

```fortran
INTEGER FUNCTION COPY(I)
IMPLICIT (INTEGER, ALL)
COMMON (CAR(3000), ORD(3000), AP(3000))
COPY = 1
IF (I .EQ. 0) RETURN
IF (AP(I) .EQ. 1) RETURN
J = 1
CALL NEW(POS)
CALL NEW(PO2)
CALL NEW(COPY)
IF (AP(CAR(J)) .EQ. 1) GOTO 2
IF (AP(CAR(J)) .EQ. 0) GOTO 2
CALL NEW(K)
CAR(ICON) = K
CALL PUSH(ICON, PO2)
CALL PUSH(J, PO2)
ICON = CAR(ICON)
J = CAR(J)
GOTO 1
2 CAR(ICON) = CAR(J)
J = FND(J)
IF (J .EQ. 0) GOTO 3
CALL APPEND(ICON, ICON)
GOTO 1
3 CALL POPUP(ICON, PO2)
CALL POPUP(J, PO2)
IF (J .EQ. 0) RETURN
J = FND(J)
IF (J .EQ. 0) GOTO 3
CALL APPEND(ICON, ICON)
GOTO 1
END
```

- 3.13. -
list processing

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.

```
5001 INTEGER FUNCTION ELEM(I1, I2)
5002 IMPLICIT INTEGER (A-H)
5003 LOGICAL I
5004 COMMON CAR(3000), LDR(3000), AF(3000)
5005 ELEM = 0
5006 I3 = I2
5007 IF(AF(I1), E0.1) GOTO 1
5009 FORMAT(1X, 'FIRST ARGUMENT OF ELEM SHOULD BE ATOM')
5011 RETURN
5012 IF(AF(I3), E0.1) GOTO 4
5014 IF(I3, EQ.2) RETURN
5016 I3 = CAR(I3)
5019 GOTO 4
5020 IF(I3, IF.1) RETURN
5022 I1 = ELEM
5023 RETURN
5024 END
```

- 3.14. -
list processing

ERASE

parameters: Il 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:

```fortran
0001  SUBROUTINE ERASE (Il)
0002  IMPLICIT INTEGER (I-N)
0003  LOGICAL *1 AF
0004  COMMON CAP(3999),CDP(3999),AF(3999)
0005  COMMON -FREE/-FREE
0006  ML = 0

C 'ERASE' REMOVES ALL CELLS USED TO REPRESENT A LIST STRUCTURE AND RETURNS
C THEM TO THE FREELIST. HOWEVER ATOMS APPEARING IN THE LIST STRUCTURE ARE NOT REMOVED.

0007  IF(AF(Il),EQ,1) RETURN
0008  IF(Il,EQ,0) RETURN
0011  CALL NEW (POS)
0012  IF(Il,EQ,2) GOTO 1
0014  TF ((AF(CAP(Il)),EQ,1),OR,(CAP(Il),EQ,0)) GOTO 2
0016  CALL PUSH (Il,POS)
0017  Il = CAP(Il)
0018  GOTO 4
0019  2  I = Il
0020  J = CAP(Il)
0021  CAP(I) = I = FREE
0022  CAP(J) = 0
0023  AF(I) = 0
0024  FREE = 1
0025  GOTO 3
0026  1  CALL POMUP (I,POS)
0027  IF(Il,EQ,1) RETURN
0029  GOTO 2
0030  END

- 3.15. -
```
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 property list of I3, I3 is set to NIL.

code:

```plaintext
SUBROUTINE GET (I1, I2, I3)

IMPLICIT none (A=0)
LOGICAL*4
COMMON CAR(3000), CPR(3000), AR(3000)

C CHECK WHETHER THE PROPERTY IS ALREADY THERE
NIL = 0
IF (AR(I1), IC, 1) GOTO 50
NITRAN(45)
50 FORMAT (45, "FIRST ARGUMENT OF GET SHOULD BE ATOM")
FALL <-
50
I1 = I1
I1 = CAR(11)
IF (CAR(11), IC, 1) GOTO 10
J1 = CAR(11)
IF (CAR(11), IC, 12) GOTO 100
C IT IS THERE
I2 = CAR(111)
RETURN
C IT IS NOT THERE
I1 = I1
RETURN
END
```

- 3.16. -
list processing

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:

```
0001 SUBROUTINE INIT
0003 C THE SUBROUTINE INIT CREATES THE FREELIST
0004 IMPLICIT INTEGER (A-H)
0005 LOGICAL 1 A
0006 COMMON CAR(3000), CDR(3000), AF(3000)
0007 COMMON JTFREE, JFREE
0009 CREATE FREELIST
0010 DO 1 I = 1, 1500
0012 AF(I) = 0
0014 CAR(I) = 0
0016 CDR(I) = I+1
0018 J = J + 1
0020 AF(J) = 0
0022 CAR(J) = 0
0024 CDR(J) = J+1
0026 IFREE = 5
0028 RETURN
0029 END
```

- 3.17. -
list processing

INPUT

parameters: IBUF, JZ, DEV

operation: INPUT is an auxiliary subroutine for the read-routines. It consumes one piece of input for the input device (DEV) starting from the IBUF-th character on the input line. A new input line 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:

0001 SUBROUTINE INPUT(IBUF, JZ, DEV)
0002 IMPLICIT INTEGER (A-H)
0003 LOGICAL * 4
0004 INTEGER ISTD(300), SCAN
0005 LOGICAL * 8
0006 INTEGER ALF(54)
0007 COMMON CAF(3000), COR(3000), AF(3000)
0008 COMMON VPRINT, VPRINT, BLANK, FIRST
0009 COMMON VSTRNV, STRIN(50)
0010 DATA ILEN/407/
0011 NIL = 0

C 1) CONTROL
C READ NEW BUFFER IF OLD ONE IS EXHAUSTED
0012 I IF(IBUF .LT. ILEN) GOTO 2
0014 IF(DEV .NE. .0) IFAD(DEV, .5, FNO=PH) (STRIN(1), I=1, ILEN)
0016 IF(DEV .NE. .0) CALL IN
0018 IF IPRTN .EQ. 1 WRITE(A, 4) (STRIN(1), I=1, ILEN)
0020 IF IPRTN .EQ. 1 FORMT (14, FRAI)
0022 IF STRIN .EQ. ALF(42) GOTO 20
0024 IF IBUF = 0
0025 IF STRIN .EQ. ALF(1) GOTO 1

- 3.18 -
```
list processing

C RECODE CHARACTER
0026 J = SCAN(TRUE)
C SEND TO VARIOUS SUBRPTS
0029 IF(J,GT,1) GOTO 4
C PUNCTUATION
0031 JZ = -J
0032 RETURN
C
C 2) ATOMS.
C (A) CHECK FOR NIL
0033 IF(1,GT,5) GOTO 12
0035 IF(STRN(TRUE+1),HE,ALF(A)) GOTO 12
0037 IF(STRN(TRUE+2),HE,ALF(2)) GOTO 12
0039 J = SCAN(TRUE+3)
0040 IF(IJ,GE,4) GOTO 12
0042 JZ = 0
0043 TRUE = TRUE+2
0044 RETURN
C
C (B) ATOMS AND HUNDEPS.
C PREPARE FOR STORING THE CODE, ATOM, AND CREATE A NEW CELL (@Z) FOR CONSULTING
C THE DICTIONARY ALSO COMPUTE THE BEGINNING OF THE DICTIONARY
0045 12 K = 1
0046 ISTRN = 1
0047 CALL NEW(I2)
0048 TD = (JU+1)
C LOOKUP BY STORING THE CODE IN I2 AND CALLING THE LOOKUP SUBROUTINE
0049 CON(I2) = 0
0050 T = CAL(I2)
0051 A = IZ
0052 CALL LOOKUP(I2,17,JZ)
0053 IF(JZ,GT,5) GOTO 14
C IF NECESSARY READ NEW BUFFER FOR NEXT CHARACTER
0055 IF(1BUL,LT,T LEN) GOTO 10
0057 IF((T E, (I2, 3) HEAD(IN, 1, ENDEP) (STRN(I1), I = 1, LEN))
0059 IF((T E, (I2, 3) CALL TD
0061 IF((T E, (I2, 1)) WHITE(6,6) (STRN(I1), I = 1, LEN))
0063 IF(STRN(I1),HE,ALF(2)) GOTO 24
0065 TRUE = 0
0066 9 TRUE = TRUE-1
C IF ELEMENT IN INPUT IS ATOM DELIMITER GOTO END OF ATOM ELSE GO ON WITH CONTI.
C NUATION OF THE DICTIONARY
0067 J = SCAN(TRUE)
0068 IF(J,LT,4) GOTO 10.
0069 K = K + 1
0071 TSTRN = 1
0072 ISTRN = 1
0074 GOTO 10
C END OF ATOM
0075 12 TRUE = TRUE-1
0076 CALL END(I2)
0077 IF(CD(R),EM,W) GOTO 120
C CHECK WHETHER BASE CELL OF ATOM WAS ALREADY IN DICTIONARY EITHER
C IMMEDIATELY AS CNK OF LAST CELL OR AS ENCODED BASE CELL
C IF NOT MAKE NEW CELL FOR EMBEDDING OR BASE CELL AND FOR PRINTNAME
0079 121 JZ = 12
0080 IF(J,NE,0) GOTO 120
0082 IF(J,GT,0) GOTO 120
0083 IF(HE,JZ,NE,W) GOTO 120
0085 IF(HE,JZ,NE,W) GOTO 120
0087 JZ = CAL(I2)
0088 RETURN
0089 127 IF(HE,JZ,82,8) GOTO 121
0091 IF(HE,JZ,82,8) GOTO 120
0093 JZ = CAL(I2)
0094 RETURN

- 3.19 -
```
list processing

C ELSE MAKE NEW CELLS

Begin

C coding for PRINTNAME

End

C (3) P-ATOMS

C (4) ERRORS AND END OF FILE

End
list processing

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:

```fortran
      LOGICAL FUNCTION LIST (I)
      LOGICAL *1 AP
      COMMON CAR(3000), CB(3000), AF(3000)
      LIST = .FALSE.
      IF(AF(I), I,N,P) LIST = .TRUE.
      END
```

- 3.21 -
list processing

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:

```plaintext
   0010  subroutine lookup(id,iz,jy)
   0020  implicit integer (a-n)
   0030  logical*1 af
   0040  common car(1000),con(3000),af(3000).
   0050  common ifree,ifree
   0060  nil = $'
   0070  c (1) consult
   0080  1 if con(id,iz,nil) goto 7
   0090  2 is = id
   0100  3 is = con(id)
   0110  4 jy = id
   0120  5 if (af(id),iz,jy) jy = con(id)
   0130  6 t = jy
   0140  7 ishown
   0150  8 if jy,iz,ln) goto 2
   0160  c (2) create new entry
   0170  9 con(id) = ifree
   0180 10 af(ifree) = id
   0190 11 con(ifree) = id
   0200 12 t = ifree
   0210 13 jy = con(id)
   0220 14 if (jy,iz,jy) goto 13
   0230 15 ifree = con(ifree)
   0240  c (3) create new cell in dictionary
   0250 16 con(id) = 12
   0260 17 t = id
   0270 18 con(id) = id
   0280 19 t = ifree
   0290 20 if (t,iz,con(id)) goto 20
   0300 21 ifree = con(id)
   0310 22 con(id) = jy
   0320 23 jy = id
   0330 24 return
   0340  c (4) return
   0350 10 write(6,11)
   0360 11 format (1x,'storage exhausted during lookup')
   0370 12 call fill - 3.22. -
   0380  end
```
list processing

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:

```
0001      SUBROUTINE NEW(I)
0002      IMPLICIT none (A-W)
0003      LOGICAL AF
0004      COMMON (S280),C280,AF(S280)
0005      COMMON IFFREE/ IFREE/
      C THE SUBROUTINE NEW TAKES ONE CELL FROM THE FREELIST
0006      IF (IFFREE .EQ. 345) GO TO 1
0007      T = IFFREE
0008      IFFREE = C280(T)
0009      C280(T) = 0
0010      WRITE(I,2)
0011      FORMAT (I, "STORAGE EXHAUSTED IN NEW")
0012      CALL EXIT
0013      END
```

- 3.23,-
list processing

PLOTLI

parameters:  II, I, K, L

operation:  PLOTLI writes a list II 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:

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

- 3.24.-
list processing

REMARKS:

1. Files from PLOT LI 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 PLOT LI, 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.
list processing

POPUP

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

operation: POPUP sets I equal to the contents of the top cell of a list II 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 II remains the same.

code:

```fortran
0001      SUBROUTINE POPUP(I,II)
0002      IMPLICIT INTEGER (A-H)
0003      LOGICAL*1 AF
0004      COMMON CAR(128),CN(3000),AF(3000)
0005      COMMON /FREE/ FREE
0006      I = CAR(I1)
0007      IF(CN(I1).GT.0) GOTO 1
0008      I2 = CN(I1)
0009      CAR(I1) = CAR(I2)
0010      CN(I1) = CN(I2)
0011      AF(I1) = AF(I2)
0012      C REMOVES SHOWN TELL
0013      CAR(I2) = FREE
0014      FREE = I2
0015      AF(I2) = I
0016      IFREE = I2
0017      RETURN
0018      CALL TELL(I1)
0019      I1 = I
0020      END
```

- 3.26. -
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 (1-5)
LOGICAL AF
COMMON CAR, COR(3000), AP(3000)
C CHECK WHETHER THE PROPERTY IS ALREADY THERE
NIL = 0
IF CAR(I1), NE, 1) GOTO 1
WRITE(6, 8)
8 FORMAT (1, 1, 'FIRST ARGUMENT OF PROP SHOULD BE ATOM')
CALL KE1
1011 J1 = I1
1012 J1 = COR(J1)
1013 IF COR(J1), NE, NIL) GOTO 14
1015 J1 = COR(J1)
1016 IF COR(CAR(J1)), NE, 12) GOTO 18
1018 IF (CAR(J1)) = 14
1019 RETURN
C IT IS NOT THERE
1020 CALL TVW(I1)
1021 CALL APEND(I1, I1, I1)
1022 COR(I1) = 12
1023 COR(I1) = 12
1024 RETURN
1025 END
```

- 3.27 -
list processing

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 output line from where the system should start printing;
if I2 is $\emptyset$ a line is left open and the system starts from the first character on the next output line.

DEV is the device on which the output must appear,
if DEV $\neq \emptyset$ the output line is constructed but not printed out. This is of use in extracting the printname of atoms via common zones.

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 output line 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

- 3.28. -
list processing

code:

```
0001  SUBROUTINE PRIST(INP,HUF,DEV)
0002  IMP (INTEGER(4*4))
0003  LOGICAL*1 AF
0004  COMMON /STRIN/STRIN(80)
0005  COMMON CAR(160),CR(160),AF(320)
0006  DATA ILEN/70/
0007
C THIS SUBROUTINE PRINTS A LIST POINTED AT BY INP ON A DEVICE CALLED DEV
C FROM THE POSITION INDICATED BY HUF
0009  NIL = 0
0010  IBUF = HUF
0011  IF (BUF,LE,1) GOTO 400
0013  CT 400: I = I,BUF
0014  401 STRIN(I) = ALF(1)
0015  402 IF ((DEV[,E,1).OR.(I,BUF[,E,1))) GOTO 402
0017  403 WRITE(DEV,403)
0018  403 FORMAT (1X/1X)
0019
0020  404 I = I,BUF
0021  405 IOUT = I
0022
0023  C TOP CONTROL : SEE WHETHER INPUT IS ATOM,NIL, OR LIST
0024  406 IF (AF(I1),EQ,1) GOTO 100
0025  407 IF (I1,LE,0) GOTO 200
0026
C IF LIST CREATE P3D CELL ON TOP OF LIST
0027  408 IOUT = 2
0028  409 I = I,1
0029  410 CALL NEP(I1)
0030  411 CAR(I1) = 1
0031  412 CALL NEP(I2)
0032  GOTO 2
0033
C NORMAL CONTROL
0034  3
0035  2
0036  1
0037  0
0038
C SECTION : PRINTING THE ATOMS
C
C GOTO PRINTNAME CELL OF ATOM, DECODE THE PRINTNAME AND WRITE IT ON THE OUT-
C PUTUFFER (STRIN)
0040  100  I = CAR(I1)
0041  101  PRN = I
0042  102  15 I = CAR(PRN)
0043  103  TREG = I,BUF = 1
0044  11 IF (BUF,L,1) GOTO 16
0045  12 IF (DEV,NE,1) WRITE(DEV,12) STRIN(I1),I=1,IHEG
0046  13 IBUF = 1
0047  14 GOTO 15
0048  15 STRING(I,BUF) = ALF(AF(I1))
0049  16 I = CAR(I1)
0050  17 IF (I,LE,1) GOTO 12
0051  18 IBUF = I,BUF = 1
0052  19 IF (I,LE,1) GOTO 16
```

- 3.29. -
list processing

0058 \texttt{STRI}(1) = \texttt{ALF}(12/12)
0059 \texttt{TUR} = 1\texttt{RU}F + 1
0060 12 = 12 = ((12/12) + 1) + 2
0061 16 \texttt{STRI}(1) = \texttt{ALF}(12)
0062 IF(C(1),1,1) GOTO 12
0064 \texttt{I} = \texttt{CO}(11)
0065 \texttt{I} = \texttt{RU}F + 1
0066 GOTO 14
C END OF ATOM ON P = ATOM / AND BLANK
0067 12 \texttt{I} = \texttt{RU}F + 1
0068 \texttt{STRI}(1) = \texttt{ALF}(1)
0069 IF(1,1,1,1) GOTO 11
0071 \texttt{I} = \texttt{RU}F + 1
0072 IF(1,1,1,1) GOTO 11
0074 GOTO 114
C
C LEFT PARENTHESES
C ---------------------------------------------------------------
C PUSH POINTERS TO CURRENT CELL AND SET CURRENT CELL EQUAL TO CAR, ADD LEFT
C PARENTHESES IF EMBEDDED NOT DUE TO AN ATOM
0075 1 IF(C(1),1,NIL) GOTO 14
0077 CALL PUS(1,4)
0078 IF(1,1,1,NIL) GOTO 114
0080 IF(1,1,1,1) GOTO 19
0082 \texttt{I} = \texttt{RU}F + 1
0083 IF(1,1,1,1) WRITE(T,6) (STRI(1),1,1,1) \texttt{I} = \texttt{RU}F + 1
0085 \texttt{I} = 1
0086 19 \texttt{STRI}(1) = \texttt{ALF}(2)
0087 \texttt{I} = \texttt{RU}F + 1
0089 117 \texttt{I} = \texttt{CAR}(1)
0091 GOTO 2
C
C RIGHT PARENTHESES
C ---------------------------------------------------------------
C POPUP POINTER TO CURRENT CELL AND ADD RIGHT PARENTHESES IF EMBEDDED IS NOT DUE TO AN ATOM, IF THE PUSHDOWN STORE IS EMPTY GOTO END
0094 114 CALL POPUP(1,4)
0091 IF(1,1,1,NIL) GOTO 14
0093 IF(1,1,1,NIL) GOTO 114
0095 IF(1,1,1,1) GOTO 22
0097 \texttt{I} = \texttt{RU}F + 1
0098 IF(1,1,1,1) WRITE(T,6) (STRI(1),1,1,1) \texttt{I} = \texttt{RU}F + 1
0100 \texttt{I} = 1
0101 22 \texttt{STRI}(1) = \texttt{ALF}(3)
0102 \texttt{I} = \texttt{RU}F + 1
0103 18 IF(1,1,1,NIL) GOTO 114
C DOT
C IF IN THE CAR THERE IS A POINTER TO AN ATOM WE ADD A DOT
0105 IF(1,1,1,NIL) GOTO 3
0107 IF(1,1,1,1) GOTO 3
0109 \texttt{I} = \texttt{RU}F + 1
0110 IF(1,1,1,1) WRITE(T,6) (STRI(1),1,1,1) \texttt{I} = \texttt{RU}F + 1
0112 \texttt{I} = 0
0113 23 \texttt{STRI}(1) = \texttt{ALF}(5)
0114 \texttt{STRI}(1) = \texttt{ALF}(1)
0115 \texttt{I} = \texttt{RU}F + 2
0116 GOTO 3

- 3.30 -
list processing

C NIL
0117 IF(IBUF.GT.0,IT,10) GOTO 210
0119 IBUF = IBUF - 1
0120 IF(DEV,NE.0) WRITE(DEV,6) (STRING(I),I=1,10)
0125 IBUF(NC) = ALF(NC)
0125 STRIN(IBUF+1) = ALF(NC)
0125 STRIN(IBUF+2) = ALF(NC)
0125 STRIN(IBUF+3) = ALF(NC)
0126 IF(IOUT.LT.1) GOTO 110
0130 GOTO 5
C END
0131 STRIN(IBUF) = ALF(3)
0132 IF(FLAG.EQ.1) RETURN
0135 IF(DEV,NE.2) WRITE(DEV,5) (STRING(I),I=1,10)
0136 IF(DEV,NE.2) WRITE(DEV,5) (STRING(I),I=1,10)
0138 RETURN
C ERROR
0140 WRITE(6,56)
0141 FORMAT (1X, 'IRREGULAR INPUT FOR PRLIST(POSSIBLY PART OF DICTIONARY *')
0142 RETURN
0143 END
list processing

PUSH

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

operation: PUSH creates a new cell on top of a list pointed at by II 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:

```
0001  SUBROUTINE PUSH(I,II)
0002  IMPLICIT INTEGER (A,N)
0003  LOGICAL*1 AF
0004  COMMON CAR(3000),DR(3000),AF(3000)
0005  COMMON IPE,PE,IPFREE, MCP
  C THIS SUBROUTINE CREATES A NEW CELL ON TOP OF A LIST II AND STORES I IN
  C THE CAR OF THIS CELL
0006  IF (II.EQ.0) GOTO 3
0007  J = I
0008  C TRANSFER INFORMATION OF FIRST CELL TO NEW CELL
0009  IF (IPE.EQ.0) GOTO 1
0010  JJ = IPE
0011  IFREE = IPE (IPE FREE)
0012  AF(JJ) = AF(J)
0013  CAR(JJ) = CAR(J)
0014  CON(JJ) = CON(J)
0015  C STORE NEW INFORMATION IN TOP CELL
0016  AF(12) = 0
0017  CAR(12) = 1
0018  COP(12) = 11
0019  II = 12
0020  RETURN
0021  1 WRITE(6,2)  C "STORAGE EXHAUSTED IN PUSH"
0022  2 FORMAT (1X, "STORAGE EXHAUSTED IN PUSH")
0023  CALL EXIT
0024  3 CALL NEW(11)
0025  GOTO 7
0026  END
```
list processing

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 $\emptyset$, 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.
list processing

3. DEV indicates the device from which the input line must be taken.
   if DEV =\emptyset a special subroutine called IN is used to fill the characters of the input line in the commonzone STRIN. The user can himself define the way in which this filling in is performed.
   If DEV is greater than \emptyset 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 input line 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 represented in 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 \emptyset).
   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
   \[ NOUN = \text{RLIST} (1,1,1) \] 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.
list processing

code:

```fortran
0001  INTEGER FUNCTION BLIST (HUF,INUF,DEV)
C
C (1) START
C
0002  IMPLICIT INTEGER (A-Z)
0003  LOGICAL IF
0004  LOGICAL IA (51), STRIN
0005  COMMON /STRING/ STRIN(50)
0006  COMMON (CAP(200),COR(300),AF(300))
0007  COMMON /PRINT/PLLIST,BLANK,FIRST
C FOR CONTROLLING THE INPUT A BUFFERPOINTER (INUF) IS USED WHICH POINTS TO THE
C FIRST CHARACTER TO BE READ. INUF INITIALLY ALSO REGULATES THE PRINTFLAG
C (IPRINT) WHICH IS SET TO 1 IF THE INPUTLINE IS TO BE PRINTED OUT, ELSE TO 0
0009  NIL = 0
0010  TRUF = HUF
0011  PLIST = 0
0012  IF (INUF.GT.1) GOTO 100
0013  IF (INUF.EQ.0) IPRINT=0
0016  IF (INUF.EQ.1) IPRINT=1
0018  TRUF = A1
C DECIDE THE FIRST INPUT ELEMENT. IF IT IS A LEFT OR RIGHT PARENTHESIS
C WE START PROCESSING FURTHER, ELSE AN ATOM IS DISCOVERED AND WE IMMEDIATELY RETURN WITH PLIST AS POINTER TO THE FIRST CELL OF THE ATOM
0019  100  CALL INPUT(INUF,JZ,DEV)
0020  IF (JZ.EQ.-1) GOTO 22
0021  IF (JZ.LT.-3) GOTO 1
0022  PLIST=JZ
0025  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
0026  1  IF (JZ.EQ.-3) GOTO 22
0028  CALL NEW(PLIST)
0029  CALL NEW(IL)
0030  IL = PLIST
0031  GOTO 11
C
C (2) MAIN PROGRAM
C
0032  7 CALL INPUT(INUF,JZ,DEV)
C
C CONTROL POINT
C
0033  IF (JZ.GT.0) GOTO 10
0035  J = JZ+1
0036  GOTO (4,3,900),J
C
C SECTION 1 ATOMS
C
C WHEN THE ATOM IS "IL", FIRST STORE = 1
0037  9 JZ = 1
```
list processing
C. IF THE CAR OF THE CURRENT CELL (IR) IS NIL, 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)
0038 10 IF(CAR(IR),EQ,NIL) GOTO 9
0040 11 IF(CAR(IR),EQ,-1) CAR(IR) = C
0042 12 CALL NEW(I)
0043 13 CAR(IR) = 1
0044 14 IR = I
0045 15 CAR(IR) = J2
0046 16 GOTO 7
C. SECTION 2, LEFT PARENTHESIS
C. WHEN THE CAR OF THE CURRENT CELL IS NOT NIL, WE FIRST CREATE A NEW CELL AND
C. HANG IT ON THE ALREADY OBTAINED LIST
0047 17 IF(CAR(IR),EQ,NIL) GOTO 6
0049 18 IF(CAR(IR),EQ,-1) CAR(IR) = 0
0051 19 CALL NEW(I)
0052 20 CAR(IR) = 1
0053 21 IR = I
0054 22 CALL PUSH(IR,I)
0055 23 CALL NEW(I)
0056 24 CAR(IR) = 1
0057 25 IR = I
0058 26 GOTO 7
C. SECTION 3, RIGHT PARENTHESIS
C. CLOSE THE LIST DOWN ( = NIL IN CAR OF CURRENT CELL) AND POPUP FROM IL
C. THE POINTER TO WHERE THE INBOXING STARTED. NOTE THE PROVISION
C. FOR NIL
0059 27 27 CAR(IR) = 0
0060 28 IF(CAR(IR),EQ,NIL) GOTO 9
0062 29 IF(CAR(IR),EQ,-1) CAR(IR) = 1
0064 30 CALL POPUP(IR,I)
0065 31 IF(CAR(IR),EQ,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
0067 32 X = LIST
0068 33 LIST = CAR(LIST)
0069 34 CALL BACK(X)
0070 35 CALL BACK(IL)
0071 36 RETURN
C. IN THE CASE OF NIL AS ( ) THE CELL DUE TO EMBEDDING IS RETURNED TO THE
C. FREE LIST AND = 1 IS STORED IN THE CAR OF THE CURRENT CELL OBTAINED BY
C. POPPING UP FROM THE PUSHDOWN
0072 37 CALL BACK(IR)
0073 38 CALL POPUP(IR,I)
0074 39 CAR(IR) = -1
0075 40 IF(IR,IF,=LIST) GOTO 7
0077 41 CALL BACK(IL)
0078 42 CALL BACK(LIST)
0079 43 RETURN
C. C(3) ERRORS
C. --------
0080 44 WRITE(I,21)
0081 45 FORMAT (IX, 'MISSING RIGHT PARENTHESIS')
0082 46 CALL EXIT
0083 47 WRITE(I,21)
0084 48 FORMAT (IX, 'MISSING LEFT PARENTHESIS')
0085 49 CALL EXIT
0086 50 WRITE(I,21)
0087 51 FORMAT (IX, 'END OF FILE DURING INPUT')
0088 52 LIST = 1
0089 53 RETURN
0090 54 END
- 3.37 -
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 hash code which guarantees fast lookup. Because we want more than 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 FDISSET(4,'UNKNOWN')
DEFINE FILE 4 (5001,41,U,IREC)
CALL ASSIGN(3,'WORD,DAT',0)
CALL FDISSET(3,'UNKNOWN')
DEFINE FILE 3(7993,17,U,IREC)

10=0
DO 100 1=1,7993
100 WRITE(4) (BL,J=1,31),10

101 WRITE(4) (BL,J=1,80)

CALL EXIT
END
```
operation:
This main program creates a dictionary by reading the atoms and storing the information about the atoms.

code:

```fortran
0001  LOGICAL *1 WORD(39), TA, WORDH(8), TAHH(2), AL
0002  LOGICAL *1 KART(AN)
0003  EUVALFMT (IA, HW(1))
0004  DATA HL/***/
0005  DATA NULL/***/
0006  CALL ASINT(5, "WORD, PAT", 0)
0007  CALL FUNKSET (5, "UNKNOWN")
0008  DEFINE FILE 4 (7003, 1, U, IREC)
0009  CALL ASINT (9, "INFO, PAT", 0)
0010  CALL FUNKSET (4, "UNKNOWN")
0011  DEFINE FILE 4 (9991, 4, U, IREC)
0012  240 READ1,90, END#200) WORD, TA
0013  99 FORMAT (41)
0014  WRITE (L, 31) IAH, INORD
0015  98 FORMAT ('V',44, A1, 53, 30A1)
0016  HW(1)=WORD(21)
0017  HW(2)=WORD(31)
0018  IAH=ORD(14, INORD)
0019  240 READ1, IA!
0020  IF (IAH(1),7941) ID=(1)
0021  READ (1, IDH, TAH, INORD)
0022  IF (WORDH(1), EQ, HL) GO TO 250
0023  DU 100 IA=1,31
0024  IF (WORDH(1), NL, WORDH(I)) GO TO 240
0025  100 CONTINUE
0026  IF (IAH(1), TAH) GO TO 240
0027  WRITE (6, 97)
0028  97 FORMAT (***
0029  94 FORMAT (***
0030  TAH=ORD(1)
0031  READ1,90) KART
0032  WRITE (6, 94) (KART(I), I=1,81)
0033  94 FORMAT (***
0034  READ (6, INORD)
0035  IF (KART(AN), EN, ML160 TO 240
0036  310 ML=I
0037  WRITE (*. INORD)
0038  IF (KART(AN), ML, ML160 TO 240
0039  INORD=TAH
0040  WRITE (*. INORD)
0041  IF (KART(AN), ML, ML160 TO 240
0042  IF (KART(I), EN, ML160 TO 240
0043  WRITE (*. INORD)
0044  IF (KART(I), ML, ML160 TO 240
0045  GO TO 240
0046  240 READ (6, INORD)
0047  GO TO 240
0048  240 READ (6, INORD)
```

- 3.41 -
parser implementation

0050  READ(1,501) INDL
0051  1NDL=INDL+1
0052  WRITE(4*INDEX) INDL, (KAART(I), I=1,80)
0053  INDEX=INDL
0054  GO TO 251
0055  250  WRITE(4*INDEX) INDEX, (KAART(I), I=1,80)
0056  INDEX=INDEX+1
0057  GO TO 215
0058  251  READ(6,501) INDEX
0059  INDEX=INDEX+1
0060  WRITE(5*INDEX), TA, INDEX
0061  251  INDEX=INDEX+1
0062  READ(1,44) KAART
0063  WRITE(6,95) (KAART(I), I=1,80)
0064  INDEX=INDEX+1
0065  IF (KAART(1), EQ, RL) INDEX=INDEX+1
0066  WRITE(4*INDEX), (KAART(I), I=1,80)
0067  IF (KAART(1), EQ, RL) GO TO 251
0068  WRITE(4*INDEX)
0069  GO TO 290
0070  300  CONTINUE
0071  END

- 3.42 -
parser implementation

SEARCH

parameters: II an atom

operation:
The integer function SEARCH consults the dictionary on
the external storage device to find the information associated
with a particular atom (II) 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:

```
0001  INTEGER FUNCTION SEARCH (II)
0002  DIM NAME(4)
0003  COMMON /IND/, INDEX
0004  COMMON /TA/ TA
0005  COMMON /ST/ STRIN
0006  DATA REF /H /
0009  CALL GFT(II,-1,SEARCH)
0101  IF(SEARCH,.NE.0) RETURN.
0102  CALL PRLIST(II,1,II)
0110  DO 1 LEN=1,5
0116  IF(STRIN(LEN).EQ.RL) GOTO 2
0117  IF 3 J = LEN39
0118  3 WORD(J) = RL
0119  INDEX(II)=(II)
0120  IF(II).NE.(II)
0121  IF(II).NE.(II)
0122  DO 4 INDEX=INDEX+1
0123  IF(II).NE.(II)
0124  WRITE(4,6)
0125  FORMAT (I15,'LINGUISTIC INFORMATION MISSING FOR :')
0126  CALL EXIT
0127  DO 992 II=1,36
0128  IF(II).NE.(II) GO TO 991
0129  CONTINUE
0130  IF(II).NE.(II) GO TO 990
0131  INDEX=INDEX
0132  SEARCH = PRLIST(J,II)
0133  CALL FPROP(II,-1,SEARCH)
0134  RETURN
0135  END
```

- 3.43 -
The subroutine fills the STRIN-vector in the commonzone for consumption by RLIST by reading items from disk. This is an auxiliary subroutine for the SEARCH operation.

```plaintext
0001 SUBROUTINE IN
0002 LOGICAL STRIN(80), RL, TEXT(80)
0003 COMMON /IND/INDX
0004 COMMON /STRIN/ STRIN
0005 DATA RL/IH /
0006 READ(*,INDX) INDX, TEXT
0007 DD I = 1, 80
0008 1 STRIN(I) = TEXT(I)
0009 RETURN
0010 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 pushdown stores 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:

```
| 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.:
parser implementation

code:

```fortran
PROGRAM PROJECT
  INTEGER FUNCTION EXT(GOAL)
  INTEGER INT(1:2)
  LOGICAL*1 AF
  COMMON: RAO(1:2), COX(1:2), AF(1:2)
  COMMON: X /1.0/, Y = 1.0, NOT
  EXT = 0
  IF (GOAL,X,NOT) RETURN

C FEATURE COMPLEX IS ATOM
  IFAR(1), X, 1) GOTO 11
  CALL X(U(1))
  CALL Y(1, X = 1)
  CALL Y(X) = 1

C FEATURE COMPLEX IN LIST
  PHASE 1
  C RECURSIVE FEATURE COMPLEX AND PUSH ON PT
  110 CALL X(1,PT)
  111 CALL X(2,PT)
  112 CALL X(1,PT)
  113 CALL X(2,PT)
  114 X = CA(1, PT)
  115 CALL PT
  116 GOTO 6

C PHASE 2
  IF (CA(PT),1,1,PT) GOTO 39
  CALL PT(1, PT)

C SEND TO RELEVANT PART
  IF (PT,1,1) GOTO 6
  IF (PT,1,1) GOTO 6
  IF (PT,1,1) GOTO 6
  IF (PT,1,1) GOTO 6

C ATOM
  CALL X(1)
  IF (X,1) GOTO 5
  CALL X(1)
  CALL X(1)
  CALL X(1)
  CALL X(1)
  CALL X(1)
  GOTO 5

C NIL
  CALL NIL(1, PT)
  GOTO 5

C NOT
  IF (X,1,1) GOTO 5
  CALL X(1)

C OR / AND
  CALL OR(1, 1)
  X = 1.1
```

- 3.47. -
CALL PEPUP(1,PP1)
0069   TF(J,EP2) GOTO 9
0062   T = CAR(PP2)
0063   TF(J,EP3) GOTO 11
0065   J1 = 1
0066   TF(J,EP4) GOTO 12
0067   F1 = CAR(J1)
0068   TF(J,EP5) GOTO 5
0069   S = COPY(F1)
0070   CALL APPEND(P,5,S)
0071   CALL APP(FT,5)
0072   T = CAR(FT)
0073   IF(J1,EP6) GOTO 13
0075   J1 = CAR(J1)
0076   IF(J1,EP7) GOTO 14
0077   CAR(PP2) = CAR(J1)
0079   CALL RACK(J1)
0080   GOTO 5
0081   123  CAR(PP2) = 1
0082   GOTO 5

C XMB
0083   113  CAR(PP2) = 1
0084   GOTO 5
0085   13   CALL POPUP(1,PP3)
0086   IF(L,EE1) GOTO 5
0088   T = CAR(PP3)
0089   21H   IF(J,EE2) GOTO 12
0090   128   CALL PUSH(PAIR(1),L)
0092   K = CAR(1)
0093   CALL RACK(1)
0094   1 = K
0095   TF(J,EE4) GOTO 7
0097   CAR(PP3) = L
0099   GOTO 5

C END
2099   Z00  CALL POPUP(EXT,PP3)
2100   CALL RACK(1)
2101   CALL RACK(PP3)
2102   RETURN
2103   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:

```
0001 INTEGER FUNCTION MATCH (SOURCE,GOAL)
0002 IMPLICIT INTEGER (A-Z)
0003 LOGICAL*1 IF
0004 COMMON CAR(380),CAR(380),AF(380)
0005 COMMON /LOGIC,OR,XOR,AND/
0006 CALL NEW(IM)
0007 IM = IM
0008 IF* = GOAL
0009 20 IFX(Element of) IX = F
0011 IFX(X,EQ.) IX = CAR(X)
0121 C PHASE (1)
0122 C DECOMPOSE FEATURE COMPLEX AND PUSH ON PD1
0133 T = SOURCE
0144 CALL NEW(PD1)
0155 CALL NEW(PD2)
0166 IFS[I,ED] = GOTO 6
0177 IFS[I,AF] = GOTO 6
0188 2 IFS[I,AF,AND,OR,XOR,AND] = GOTO 1
0222 CALL PUSH(PD1,PD2)
0233 T = CAR(T)
0244 GOTO 2
```
parser implementation

0025 1 CALL PUSH(color(1,PM1)
0026 20 I = COLOR(1)
0027 3 IF (1,EQ,0) GOTO 10
0029 5 IF (CAR(PO2),EQ,2) GOTO 5
0031 7 CALL PURPLE(1,PM1)
0032 10 GOTO 22
0033 11 CALL PUSH(1,PM1)
0034 9 C PHASE(2)
0034 9 IF (COLOR(1,PM1),EQ,9) GOTO 9
0036 12 CALL PURPLE(1,PM1)
0037 13 C SEND TO RELEVANT PARTS
0039 14 IF (I,EQ,0) GOTO 9
0039 15 IF (I,EQ,9) GOTO 14
0040 16 IF (I,EQ,9) GOTO 15
0041 17 IF (I,EQ,9) GOTO 12
0044 18 IF (I,EQ,9) GOTO 13
0047 19 C ATOMS
0048 19 CALL PUSH(MATCH1 (1,1K),PR2)
0050 20 GOTO 6
0051 21 C NIL
0052 22 CALL PUSH(1,PR2)
0054 23 GOTO 6
0055 24 C NOT
0056 25 IF (CAR(PO2).EQ,0) GOTO 60
0058 26 CAR(PO2) = CAR(PO2) + 1
0060 27 GOTO 6
0055 28 C OR
0056 29 IF (CAR(PO2),EQ,4) GOTO 67
0057 30 CALL PURPLE(1,PO2)
0058 31 IF (I,EQ,1) CAR(PO2) = 1
0060 32 GOTO 6
0061 33 C AND
0062 34 IF (CAR(PO2),EQ,2) GOTO 60
0063 35 CALL PURPLE(1,PO2)
0064 36 IF (I,EQ,1) CAR(PO2) = 1
0066 37 GOTO 6
0067 38 C NOR
0068 39 IF (CAR(PO2),EQ,0) GOTO 60
0069 40 CALL PURPLE(1,PO2)
0070 41 IF (I,EQ,1) GOTO 34
0072 42 IF (CAR(PO2),EQ,4) GOTO 67
0073 43 CAR(PO2) = -1
0075 44 GOTO 6
0076 45 IF (CAR(PO2),EQ,2) GOTO 60
0077 46 CAR(PO2) = 1
0079 47 GOTO 6
0080 48 C ITE(6,41)
0081 49 FORMAT (1X, 'UNRECOGNIZED FEATURE COMBINATION IN MATCH TEST')
0082 50 MATCH = -1
0083 51 GOTO 31
0084 52 IF (CAR(PO2),EQ,0) GOTO 60
0086 53 CALL PURPLE(1,PM2)
0087 54 IF (CAR(PO2),EQ,1) GOTO 67
0088 55 C ACCUMULATE RESULTS AND END
0089 56 CALL MATCH(PO2)
0090 57 CALL MATCH(PO1)
0091 58 IF (MATCH,EQ,1) CALL APPEND (1M,1K,IM)
0093 59 K = CAR(1M)
0094 60 IF (I,EQ,1) GOTO 20
0096 61 MATCH = 3
0097 62 IF (CAR(1,1K),EQ,0) RETURN
0099 63 MATCH = CAR(0)
0100 64 CALL MATCH(1M)
0101 65 RETURN
0102 66 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:

```
2001 INTEGER FUNCTION MATCH2(J,IK,INFTR)
2002 IMPLICIT INTEGER (4)
2003 LOGICAL*1 IF
2004 COMMON CAN(1009),CAR(309),AF(1009)
2005 MATCH2=1
2006 K = IK
2007 IF(K.EQ.2) GOTO 11
2008 IF(CAR(K).EQ.J) GOTO 10
2011 IF(INFTR.EQ.1) GOTO 2
2013 IF(CAR(K).EQ.J) GOTO 10
2015 K = CAR(K)
2016 GOTO 1
2017 IF(MATCH2.EQ.1) GOTO 11
2018 RETURN
2019 IF(J.EQ.3) RETURN
2021 GOTO 19
2022 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 pushdown store (PDS) and by setting flags at relevant points during scanning.

code:

```fortran
0001  INTEGER FUNCTION CROSS(SOU, GOAL, INFTR)
0002  IMPLICIT INTEGER(I-H)
0003  LOGICAL=1 AF
0004  COMMON CAR(3000), CDR(3000), AF(3000)
0005  CROSS = 0
0006  CALL NEW(PDS)
0007  CALL NEW(LI)
0008  S = LI
0009  CALL POPUP(LI, PDS)
0010   IF(CAR(LI), EQ, SOU) GOTO 2
0011   IF(CAR(LI), EQ, GOAL) GOTO 2
0012   IF(CARR(LI), EQ, SOU) GOTO 2
0013   IF(CARR(LI), EQ, GOAL) GOTO 2
0014   IF(CARR(LI), NE, S) GOTO 3
0015   IF(CARR(LI), NE, S) GOTO 4
0016   CALL RACK(S)
0017   RETURN
0018   CALL PUSH(LI, PDS)
0019   LI = CAR(LI)
0020   GOTO 3
0021   CALL POPUP(LI, PDS)
0022   IF(LI, EQ, P) GOTO 6
0023   IF(CARR(CARR(LI)), EQ, GOAL) GOTO 5
0024   GOTO 2
0025   CALL ERASE(PDS)
0026   CALL RACK(S)
0027   CROSS = 1
0028   RETURN
0029   END
```

- 3.52 -
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-H)
LOGICAL * 1, AF
COMMON (CAR(3), CAR(3)), AF(3)
COMS = I1
IF (J1.EQ.0) RETURN
COMS = J1
IF (I1.EQ.0) RETURN
CALL NEW (COMS)
C = COMS
IF = C
J = J1
IF (J1.EQ.0) GOTO 2
IF = I1
IF (I1.EQ.0) GOTO 3
F = COPY (CAR(1))
CALL ADD (CAR(J), F)
CALL APPEND (CAR(1), C)
I = CAR(1)
GOTO 1
GOTO 0
J = CAR(J)
GOTO 1
GOTO 2
COMS = CAR(IF)
CALL RACK (IF)
RETURN
END
```

- 3.53. π
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: \((a_1, a_2, a_3, a_4)\) where \(a_1\) is the start state of a transition, \(a_2\) is the resulting state, \(a_3\) is the condition for the transition to take place and \(a_4\) is the symbol associated with the transition.

- \(a_1\) may be one state or a feature complex of states
- \(a_2\) may be one state or a list of states
- \(a_3\) is a feature complex containing features
- \(a_4\) 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 \((\text{CON}, \text{STAT}, \text{RES})\) where

- \(\text{CON}\) denotes the condition for a transition to take place (\(\text{CON}\) is the extension of a feature complex)
- \(\text{STAT}\) denotes a state (or a set of states)
- \(\text{RES}\) denotes possibly a symbol associated with the transition.

The idea is that if \(\text{CON}\) is NIL, \(\text{RES}\) 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 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) \((b_1,b_2,b_3)\) with
\(b_1\) the resulting domain of the conditional feature complex
\(b_2\) a new state (or a set of new states)
\(b_3\) 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 straightforward 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:
PARSER IMPLEMENTATION

INTEGER FUNCTION NETN(CON, STAT, RES, NF, NF, NF)
IMPLICIT INTEGER (A-H)
LOGICAL*1 AF
COMMON CDR(31, 30), CRD(31, 30), AF(30)
NETN(0)
FLA = IF(CON.EQ.0) FLA = 0
IF ((FLA.EQ.1).AND.(RES.EQ.0)) RETURN
INET = INET
CALL NFNF(NF)
S = OUTP
CALL NEWTIS
C CHECK WHETHER CONDITION IS SATISFIED
1 IF(INET.EQ.1) GOTO 10
IF(RES.EQ.0) GOTO 5
IF ((FLA.EQ.1)) GOTO 5
TRES = MATCH (CAR(CDR(CAR(INET)))). CON, FUNTK)
CALL PLLST (TRES, 1, 6)
CALL PLLST (CON, 1, 6)
IF (INET.EQ.0) GOTO 15
GOTO 24
IF (RES.EQ.0) GOTO 15
C CHECK WHETHER STATE IS SATISFIED
20 NSFAT = MATCH (CAR(NET)), IS, INT
28 CALL PLLST (NSAT, 1, 6)
29 CALL PLLST (IS, 1, 6)
30 IF (NSAT.EQ.0) GOTO 15
C ADD NEW TRIPLE TO OUTPUT
32 CALL NFNE(l)
33 CALL APPEND (S, L, S)
34 CALL APPEND (L, CAR(CON(CAR(INET)))). I)
35 IF ((CON(CAR(INET))). NE.0)
* CALL APPEND (1, CAR(CDR(CAR(CAR(INET))))). 1)
36 INET = CAR(INET)
37 GOTO 1
C END
10 IF (CON(NF).EQ.0) GOTO 11
11 CALL BACK(NF)
RETURN
14 I = CON(NF)
15 CALL BACK(NF)
NETNF
OUTP = I
RETURN
END
- 3.56.-
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.
IMPLICIT INTEGER (1-X)

LOGICAL * 1 TA
LOGICAL * 1 AF
COMMON /IFREP/IFREP
COMMON /VECT/VECT(30),VRMPS
COMMON /RETF,SYNTH,SEMTRE,FUNKRE
COMMON /MONSYNTH,VERBAL,CASE1

COMMON C100/ CODE / LOCK, RULE, BEFORE, AFTER, THUE, FALSE, UNDEF, PREDIC
COMMON C100/ MIN, DUAL, ANU
COMMON /INVER/ INVER, METATE, IN
COMMON /LOG / AND, OR, NOT
COMMON /CONF / COMP (3P, 1D)
COMMON /ETH/ FIN, TR
COMMON /TA/ TA
COMMON /POS/ POS, PUN?
COMMON // VEF
COMMON CAR (3000), COR (3000), AF (3000)
CALL INIT
C READ SYMALS
CALL ASSIGN (3, 'WORD, DAT*, 0)
CALL FORSET (3, 'UNKNOWN')
DEFINE FILE 1 (1993, IT, U, IREC)
CALL ASSIGN (4, 'INFO, DAT*, 0)
CALL FORSET (4, 'UNKNOWN')
DEFINE FILE 1 (1991, 01, U, IREC)
TR = P
NIL = 2
CALL NEW (POS)
CALL NEW (POS)
CODES = BLIST (0, 1, 2)
CODE = CODE
WORD = CAR (CODE)
LOCK = CAR (COR (CODE))
RULE = CAR (CAR (CAR (COD)));
REFFRE = CAR (CAR (CAR (CAR (CODE))))
AFTER = CAR (CAR (CAR (CAR (CAR (CODE)))))
ICODF = CAR (CAR (CAR (CAR (CAR (CAR (CODE)))))
THFRE = CAR (ICODF)
UNDEF = CAR (UNDEF)
AND = CAR (CAR (CAR (ICODF)))
FUNCT = CAR (CAR (CAR (CAR (ICODF))))
OBJE = CAR (CAR (CAR (CAR (COR (ICODF)))))
INDE = CAR (CAR (CAR (CAR (COR (ICODF)))))
FRAME = CAR (ICODF)
SYNTH = CAR (COR (ICODF))
AND = CAR (COR (COR (ICODF)))
OR = CAR (COR (COR (ICODF)))
AND = CAR (CAR (CAR (CAR (ICODF))))
ICODF = CAR (CAR (CAR (CAR (COR (ICODF))))
FIN = CAR (ICODF)
C READ INFERENCE TABLES
0116 29 SYNTRE = SEARCH (SYNT)
0117 29 SYMR = SEARCH (SYMP)
0118 29 FINTRE = SEARCH (FUNCTION)
0119 28 CALL NEX (INSTAT)
0120 28 CALL NEX (INSTAT)
0121 30 WRITE (4, I022)
0122 1002 FORMAT (1X, 'GIVE INPUT SENTENCE')
0123 1002 WORDS = 1
C SENTENCE COMES IN
0125 1002 INP = BLIST (0, I, 1)
0126 1002 IF (INP.EQ.0) GOTO 450
0127 1002 I = INP
0128 1002 IF (INP.EQ.TRACE) TP = 1
0129 1002 IF (INP.EQ.UNDOD) TH = 0
0130 1002 IF (INP.EQ.TRACE).OR.(INP.EQ.UNDOD) GOTO 49
0131 1002 INP = INP
0132 1002 IF (INP.EQ.0) GOTO 550
0133 35 J = SEARCH (CAB (1))
0134 1002 IF (CAB (1).EQ.0) GOTO 40
0135 1002 I = FOR (1)
0136 1002 GOTO 35
0137 40 CONTINUE
0138 1003 WRITE (4, 1003)
0139 1003 FORMAT ('1X, 'IN:')
0140 1003 CALL PLIST (INP, I, A)
0141 1003 IF (INP.EQ.1) WRITE (4, 1004)
0142 1004 FORMAT (1X, 'CONFIGURATIONS IN THE STATESPACE:')
0143 1004 TP = 1
C TAKE NEW WORD
0150 35 WORD = CAB (INPP)
0151 1004 FORMAT (1X, 'NEW WORD')
0152 1004 IF (TP.EQ.1) WRITE (4, 1004)
0153 1004 CALL PLIST (WORD, 1, A)
0154 1004 IF (TP.EQ.1) WRITE (4, 1004)
0155 1004 IF (TP.EQ.1) WRITE (4, 1004)
0156 1004 IF (TP.EQ.1) WRITE (4, 1004)
0157 1004 IF (TP.EQ.1) WRITE (4, 1004)
C C CONSTRUCT INITIAL STRUCTURE
0158 1004 FORMAT (1X, 'INITIAL PARTICLES')
0159 1004 CALL NEX (K)
0160 1004 MLIST = K
0161 1004 CAR (K) = WORD
0162 1004 CALL NEW (L)
0163 1004 CALL APREM (K, L, K)
0164 1004 CAR (L) = CAR (HYPL)
0165 1004 ON = L
C FOR EACH LEXICAL INFORMATION LINE CONSTRUCT PARTICLE
- 3.60 -
parser implementation

0144  FUNC = CAR(CAR(FIMRF))
0147  TFUN = 0
0148  IF(AF(FUNC),EQ,.1) GOTO 4
0149  IFUN = FUNC
0171  2  FUNC = CAR(IFUN)
0172  IFUN = CDW(IFUN)
0173  32  TF (FLAN,FG,1) GOTO 3
0175  FLAG = 1
0176  GOTO 9
0177  3  CALL NEW(L)
0178  CALL APPEND (L, CAR(L), K)
0179  CAR(L) = CAR(NYPL)
0180  ON = L
0181  HYPL = CAR(NYPL)
0182  CALL PROFWNU, HYPL, L)
0183  HYPL = CDW(NYPL)
0184  4  CALL APPEND (L, CAR(IMORE), L)
0185  CALL NEW(A)
0186  CALL APPEND (L, F, L)
0187  IF(WORDS, F, L) GOTO 5
0188  CALL PUSH(L, INVRS)
0189  5  CALL IFUN(IF)
0191  CAR(F) = 1
0192  CALL APPEND (F, WORD-1, F)
0193  DIF (F) = 0
0194  CALL GET(IFUN,RULE, L)
0195  TF (IN, ELD, L) GOTO 55a
0197  NNET = 2
0198  ANET = 0
0199  CALL GET (FUNC,BEFORE, NNET)
0200  CALL GET (FUNC, AFTER, ANET)
0201  CALL GET (FUNC, WORD)
0202  IF (WORDS, NE, 1, AND, NNET, NE, 0) CAR(J) = CAR(NWFT)
0203  CALL IMPJ(J)
0204  CALL IMPJ(J)
0205  CALL APPEND (J, I, N, J)
0206  CALL IMPJ(J)
0207  CALL APPEND (I, FUNC, I)
0208  CALL APPEND (I, O, J)
0209  IF (ANET, NE, 0) CAR(J) = CAR(ANET)
0210  J = J
0211  C (4) STATE IN CASE NETWORK (UNKNOWN YET)
0212  CALL APPEND (I, O, J)
0213  CALL APPEND (I, O, J)
0214  CALL APPEND (I, O, J)
0215  IF (IN, ELD, L) GOTO 5
0216  IF (IN, ELD, L) GOTO 9
0217  IF (AFL, F, L, DP, CAR(I4), ED, NOT, DP, CAR(I8), FG,)

- 3.61 -
PARSER IMPLEMENTATION

1 AND OR CAR(J4), EQ, OR, OR, CAR(J4), EQ, OR) GOTO 4

2 CAP(1) = EXT(CAR(CONF(J4)))
3 GOTO 0

C OBJECTS
3 (5) SYNT FEAT COMPLEX
027 6 J = EXT(CAR(CDR(CDR(CDR(CAR(IMORP))))))
028 CALL APPEND (1, J, 1)

C (6) SEM FEAT COMPLEX
029 CASE = CAR(CDR(CDR(CAR(IMORP))))
030 J = S(EA.M(CAR(CDR(CAR(IMORP))))
031 IF(CAP(CAP(1)), EP, CASE) GOTO 8
033 J = CAR(J)
034 IF(J, NE, 81 GOTO 7
035 WRITE(8, 1904)
037 1000 FORMAT (1X, "MISSING CASE IN FRAME ")
039 GOTO 94

C (7) CASE (UNKNOWN YET EXCEPT FOR ADJUNCTIVE OBJECTS)
040 CALL APPEND (1, EXT(CAR(CONF(J)))

7

C (START PARSE
041 IF(CONV(HYP), EQ, 8) GOTO 10
042 TH = CONV(HYP)
043 GOTO 50

C

C COMPUTE SEMANTIC STRUCTURES
050 A FINL = VECT(WORDS)
051 HYP = HYP
052 T = 0
054 WRITE (6, 440)
055 FORMAT (1X/4, "FUNCTIONAL AND CASE STRUCTURES : ")
056 CALL CLOSE(4)
057 93 FINL = CAR(FINL)
058 CALL CAR(CAR(HYP))
059 CONV = CAR(CAR(HYP))
060 IF(CAP(CAR(CAR(IMORP))), NE, 8) GOTO 94
061 T = CONV(CAR(CAR(CONF)))
063 CALL FUND(T)
064 IF(1, NE, 8) GOTO 90
065 T = T+1
067 CALL CASE(T)
068 90 CONV = CAR(CONV)
069 IF(CONV, EQ, 8) GOTO 91
070 GOTO 92
071 IF(FINL, EQ, 8) GOTO 93
072 IF(T, NE, P) WRITE (6, 556)
073 556 FORMAT (1X, "NO STRUCTURE FOR GIVEN INPUT")
074 90 CONTINUE
080 TH = 0
081 WRITE (6, 555) 5500. IFPEF
085 555 FORMAT (1X/4, "MEMORY CELLS LEFT : 14")
086 CALL CLOSE(4)
087 CALL ASSIGN(6, "INFO.DAT", 0)
088 CALL FORSET(6, "UNKNOWN")
089 DEFINE FILE 4 (B981, 41, U, REC)
090 GOTO 10
091 CONTINUE
092 END

- 3.62 -
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-Z)
LOGICAL = (L, 10)
LOGICAL = (L, 10)
COMMON CAR(3999), CAR(3999), AF(3999)
COMMON STACK(39, 17)
COMMON CODE, LOCK, RULE, BEFORE, AFTER, TRUE, FALSE, UNDEF, FUNCTION
COMMON INVES, INVES, NSTATS, LD
COMMON /VECT/ VECT(3A), WORDS
COMMON /VERB/ VERB
COMMON /POS, POS, POSP
COMMON /FREE/ IFREE
DATA ALF, AF, A, E, P, G, H, T, J, J
N = NSTATS
C TAKE TASK FROM TASKLIST
IF(CAR(INVES), EQ, 0) GOTO 19
CALL POPUP(CONF, INVES)
STRUCT = CAR(CONF)
OWNR = CAR(CDR(CONF))
A = A+1
WRITE(6, 171) ALF(A)
1911 FORMAT (1X, '(', A1, ')')
WRITE(6, 170)
190 FORMAT (1X, '**** TRY TO EXPAND CONFIGURATION ****')
CALL PRLIST(STRUCT, 5, 6)
CHYL = VECT(OWNR)
WRITE(6, 172) (NOK)
1722 FORMAT ('** BY COMBINING IT WITH CONF OF WORD NR. *, I3')
T = T1
C GET PARTICLES ORDERING ON INVESTIGATED CONFIG
2 CHYP = CAR(CHYL)
T1 = T1 + 1
CALL PRLIST(CAR(UNCR), 2P, A)
WRITE(6, 187) T1
187 FORMAT (1H*, IP, *, FOR HYPOTHESIS ***)
T = T
PCONF = CAR(CDR(CHYP))
2 3 203 PCONF = CAR(0CONF)
204 IF(CAR(0CONF), EQ, LOCK) GOTO 100
205 T = CAR(CDR(CAR(CDR(CAR(0CONF))))))
206 J = CAR(CDR(CDR(CAR(0CONF)))
207 IF(I, EQ, VERB, AND, J, EQ, VERB) GOTO 109
T2 = T2 + 1
WRITE(6, 103) T1, T2
103 FORMAT (3X, IP, *, IP, *, * CONFIGURATION **)
CALL PRLIST(CAR(0CONF), 4, 6)
T3 = T
WRITE(6, 104) T1, T2
104 FORMAT (3X, ** FROM LEFT TO RIGHT **)
WRITE(6, 105)
105 FORMAT (3X, ** FROM LEFT TO RIGHT **)
106 CALL LINGUISTIC PROCESSOR FOR LEFT TO RIGHT COMBINATION
107 CALL LINGUISTIC PROCESSOR FOR RIGHT TO LEFT COMBINATION
C FOR EACH "RIGHTMOST NODE" IN THE STRUCTURE
parser implementation

D WRITE(A145)
D0034 FORMAT (A) REM FROM RIGHT TO LEFT)
D0035 I = CDRI(CAR(ODCF)}
D0036 P01N = 1
D0037 291 T = CDRI(I)
D0038 IF(COR(I),NE,0) GOTO 191
D0039 CALL P01ST(CAR(POIN),P9,A)
D0040 T3 = T3 + 1
D0041 WRITE(A,10A) T1,T2,T3
D198 FORMAT (1X,TX,12,",",IP,"",IP,"",IP,"",IP, FOR WORD 1)
D0042 IF(CAR(POS1),FL,0) GOTO 196
D0043 CALL PO0UP(T1,POS)
D0044 CALL POPUP(POIN,POS2)
D0045 GOTO 200
D0046 196 IF(CAR(COR(I)),EQ,0) GOTO 197
D0047 T = COR(I)
D0048 CALL PUSH(T,PDS)
D0049 CALL PUSH(POIN,POS2)
D0050 T = CAR(I)
D0051 POIN = 1
D0052 G0T0 201
D0053 199 IF(COR(ODCF),NE,0) GOTO 202
D0054 ODCF = COR(ODCF)
D0055 GOTO 203
D0056 202 CONTINUE
D0057 3 IF(COR(NHYP),EQ,0) GOTO 1
D0058 NHYP = COR(NHYP)
D0059 GOTO 2
C ATTACH RESULTING PANTICLES AND LOCK
D0060 19 NSTAT = S
D0061 12 IF(CAR(NSTAT),EQ,0) GOTO 13
D0062 CALL POPUP(J,NSTAT)
D0063 CONF = J
D0064 NHYP = COR(CONF)
D0065 J = COR(NHYP)
D0066 IF(COR(J),NE,0) GOTO 11
D0067 CALL APPEND (I,J,J)
D0068 GOTO 12
D0069 13 IF(CAR(LO),EQ,0) RETURN
D0070 14 CALL POPUP(I,LO)
D0071 CAR(CAR(1)) = LOCK
D0072 GOTO 13
D0073 END

- 3.65. -
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:
(11) predicates for objects:

\[
\text{p-taking-objects} \quad \text{p-object-position} \quad \text{p-semfeat.objects} \quad \text{p-sem.netw}
\]

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:

```
if p-position
  true
  false
if p-synt-net
  true
  false
if p-function-of
  false
head
  true
  false
if p-concord
  false
true
if p-sem.feature
  true
true
MERGE
```

parser implementation

for objects:

```
if p-object-position
    true
    false

if p-taking-objects
    false
    true
    if p-sem-net
        true
        false
        if p-semfeat
            true
            MERGE
            NO 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.

(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 and function. 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 semantic 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 L0(NCONF,CDEF,F,POINT)
IMPLICIT INTEGER (4,7)
LOGICAL*1 AF
COMMON CAR(300),COR(300),AF(300)
COMMON LOG,AND,OR,XOR,NOT
COMMON /SYMP,SYN,TRE,SENT,FINTRK
COMMON /CONF/ CONF(30,10)
COMMON /CODE/ LUCK,RULE,REF0RE,AF0ER,TRUE,FLASE,UNDEF,FUNCT,
  SYM,T,HR,OBJC,UNPL,PREDIC
COMMON /INVS/,INVE,NSSTAT,LO
COMMON /FIN/ FIN,TR
COMMON /IFREE(IFREE
COMMON /MOD/ MOD,QUAL,ANJU
COMMON /AD/ SYN,T,VERB,CASE1
C INITIALIZE CHANGE INDICATORS
0014 DBFM = 0
0015 ANES = 0
0016 RES = 0
0017 GSYN = 0
0018 T = 0
0019 OUTP = 0
0020 TACE = 0
0021 NFM = 0
0022 NSYN = 0
0023 CHAP = 0
0024 NEWS = 0
0025 CASEF = 0
0026 SYMP = 0
0027 DU = 0
0028 NRES = 0
0029 DU = 1
C GET RELEVANT INFORMATION POINTERS
0030 NSTRUC = CAR(COR(NCONF))
0031 STRUCT = CAR(COR CONF)
0032 CDRUC = CAR(COR(NCONF))
0033 CALL GET(CAR(OSTRUC),CAR(COR(CDRUC))),NHYP)
0034 CALL GETCAR(POINT),CAR(COR(CDR(POINT))),NHYP)
C GET INFORMATION (O/N/FEAT)
0035 OPER = CAR(NCONF))
0036 NFEAT = CAR(COR(NCONF))
C GET INFORMATION SEQUENCE (O/N/INF)
0037 NINF = CAR(COR(NCONF))
C GET FUNCTION (O/N/FUNC)
0038 NFUNC = CAR(COR(NCONF))
0039 NINF = CAR(COR(NCONF))
C GET SYNTACTIC RULE (O/N/RULE)
0040 CALL GET(NFUNC,RULE,NRULE)
0041 CALL GET(NFUNC,RULE,ORULE)
C (A) NETWORKS
C (A) GET NETWORK
0042 IF(F,EQ,1) CALL GET(NFUNC,REFORE,NNET)
0043 IF(F,EQ,1) CALL GET(NFUNC,AFTER,NNET)
0044 IF(NNET,EQ,1) GOTO 2
C (A) GET STATE
0045 IF(F,EQ,1) NSTATE =CAR(CAR(NCONF))
0046 IF(F,EQ,1) NSTATE = CAR(COR(COR(NCONF)))
- 3.73 -
parser implementation

0042   IF (STATE.EQ.2) STATE = CAR (NNET)
0044   INFTR = CAR (COND (NNET))
0045   NNET = CAR (CDR (COND (NNET))))

C (c) PREPARE INPUT FOR NETW
0046   CALL NEW (COND)
0047   CALL NEW (T)
0048   CAR (T) = OFUNC
0049   CAR (COND) = T
0050   L = 0
0051   J = STATE

C (d) CONSULT
0052   T = NETW (COND, STATE, L, K, NNET, INFTR, FUNTRE)
0053   CALL ERASF (COND)
0054   TF (I.EQ.0) GOTO 2

C (e) FILTER
0055   CALL NEW (NEWS)
0056   L = NEWS
0057   I = K
0058   TF (I.EQ.11) GOTO 12
0059   CALL AND (CAR (COND) (CAR (I) 1), NEWS)
0060   T = COND (I)
0061   GOTO 11

0062   11 NEWS = COND (NEWS)
0063   CALL BACK (I)
0064   CALL PRIST (J, 35, 6)
0065   WRITE (6, 100)
0066   100 FORMAT (I4, 7X, "SUCCESSFUL TRANSITION FROM")
0067   CALL PRIST (NEWS, 31, 6)
0068   300 FORMAT (I4, 7X, "TO THE NEW STATE(S) :) ")
0069   IF (F.EQ.1) NEWS = NEWS
0070   IF ( F.EQ.1 ) NEWS = 0
0071   GOTO 3

C (2) FUNCTION OF HEAD/POSITION
0072   IF (STATE.EQ.0) POS = 0
0073   IF (CAR (FORULE, 3).EQ. ORDER) POS = CAR (FORULE, 6)
0074   IF (CAR (FORULE, 3).EQ. OBJECT) POS = CAR (FORULE, 5)
0075   IF (POS.EQ.0) GOTO 1001
0076   IF (F.EQ.9) AND POS.EQ. AFTER) GOTO 1091
0077   IF (F.EQ.9) AND POS.EQ. BEFORE) GOTO 1091
0078   CALL NEW (COND)
0079   CALL NEVF (T)
0080   IF (I.EQ.1) GOTO 1091
0081   IF (I.EQ.1) WRITE (6, 101)
0082   101 FORMAT (94, " SUCCESSFUL ORDER AND RELATIONS ENVIRONMENT TESTS")

C (3) SYNT FEATURES
0083   3 IF (CAR (FORULE, 3).EQ. ORDER) GOTO 6
0084   IF (CAR (FORULE, 7).EQ. TRUE) GOTO 35

C (1) GET FEATURES
0085   NDIF = CAR (CDR (CDR (COND (COND (NNET))))))
0086   OFEAS = CAR (CDR (CDR (COND (CDR (OFEAS))))))
0087   TF (IF (OFEAS, EQ, 11) GOTO 31
0088   IF (CAR (OFEAS).EQ.0) AND. OR. CAR (OFEAS) = .EQ.0 AND. OR. CAR (OFEAS)

3.74.
0117  nFEAS = CAR(OFFSET)
0118 31  CONTINUE
0119  (III) MATCHING
0120  N  WRITE (6,103)
0121  0103  FORMAT (6x,"MATCH THE FOLLOWING FEATURE COMPLEXES;")
0122  N  CALL PRLIST (nFEAS;A,6)
0123  N  CALL PRLIST (nDOM;A,6)
0124  0112  RES = MATCH (nFEAS, nDOM, SYNTH)
0125  0113  IF (RES.EQ.3) GOTO 1012
0126  0115  44  CONTINUE
0127  N  WRITE (6,102)
0128  0102  FORMAT (6x,"RESULTING DOMAIN")
0129  N  CALL PRLIST (RES;A,6)
0130  C (III) SEND-THROUGH
0131  0117  35  IF (CONF.(RULF;R), NE, TRUE) GOTO 4
0132  0119  IF (RES. NE, 3) RES = COPY(RES)
0133  0121  IF (RPS.EQ.0) RES = CAR(COR(COR(COR(INF))))
0134  0123  NSYN = COR. TEXT(CAR(COR(COR(COR(COR(OFAT))))))
0135  N  WRITE (6,101)
0136  0106  FORMAT (6x,"NEW FEATURE COMPLEX:")
0137  N  CALL PRLIST (NSYN,A,6)
0138  C (4) SEMANTIC FEATURES TEST
0139  0124  6  IF (CONF.(RULF;R), EQ, 3) GOTO 5
0140  C (II) SEARCH INFORMATION SEQUENCES
0141  0126  INFEAT = NFEAT
0142  0127  TININF = NINF
0143  0128  TINRULE = NRULF
0144  0129  IF (OPFX.NE, VERBAL) GOTO 31
0145  0131  SUBJ = CAR(COR(COR(COR(STRUCT))))
0146  0132  IF (CAR(COR(COR(INF))), NE, TRUE) GOTO 1043
0147  0134  CALL GET (CAR(SUBJ), CAR(CAR(COR(SURF))))
0148  0135  INFEAT = CAR(COR(INFHY))
0149  0136  TININF = CAR(COR(SUBJ))
0150  0137  CALL GET (CAR(INFEAT), RULE, RULE)
0151  0138  41  I = 0
0152  0139  IF (CONF.(RULE;R), NE, OBJECT) I =
0153  E CAR(COR(COR(COR(COR(INF))))))
0154  0141  STYP = CONF.(RULF,R)
0155  0142  NRES = FRAMES (INFEAT, OFAT, STYP, I)
0156  0143  IF (NRES.EQ, 4) GOTO 1043
0157  N  WRITE (6,107)
0158  0107  FORMAT (6x,"SEMANTIC FEATURES MATCH SUCCESSFUL, DOMAIN ":)
0159  N  CALL PRLIST (NRES;A,6)
0160  0145  IN = 1
0161  0146  GOTO 5
0162  C (II) OBJECT
0163  C (II) SEMANTIC NETWORKS FOR SURFACE CASE SIGNALS
0164  0147  6  ROLES = SEMANT (CAR(COR(NFEAT))
0165  0148  3ROLE = CAR(COR(COR(COR(NFEAT))))
0166  0149  CALL NEW (NFUN)
0167  0150  CALL NEW (1)
0168  0151  CAR(NFUNS) = I
0169  0152  CAR (1) = NFUN
0170  0153  61  IF (CAR(COR(ROLES), EQ, 3ROLE) GOTO 62

- 3.75. -
<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0155</td>
<td>ROLES = CAR(ROLES)</td>
</tr>
<tr>
<td>0156</td>
<td>IF (ROLES =0,WHILE) GOTO 62</td>
</tr>
<tr>
<td>0158</td>
<td>IF (ROLES =0,WHILE) GOTO 105</td>
</tr>
<tr>
<td>0160</td>
<td>GOTO 61</td>
</tr>
<tr>
<td>0161</td>
<td>62 ASSO = CAR(CAR(ROLES))</td>
</tr>
<tr>
<td>0162</td>
<td>IF (ASSO =0,WHILE) GOTO 105</td>
</tr>
<tr>
<td>0164</td>
<td>63 IF (MATCH(CAR(CAR(ASSO)),IFUNC,FUNCT),NF,9) GOTO 44</td>
</tr>
<tr>
<td>0166</td>
<td>ASSO = CAR(ROLES)</td>
</tr>
<tr>
<td>0167</td>
<td>IF (ASSO =0,WHILE) GOTO 105</td>
</tr>
<tr>
<td>0169</td>
<td>GOTO 63</td>
</tr>
<tr>
<td>0170</td>
<td>64 INF = CAR(ROLES)</td>
</tr>
<tr>
<td>0171</td>
<td>IF (INF =0,WHILE) GOTO 105</td>
</tr>
<tr>
<td>0172</td>
<td>WRITE (6,109)</td>
</tr>
<tr>
<td>0173</td>
<td>FORMAT (AX, 'CONSULT CASE FRAMES WITH SYNT FEATURES :')</td>
</tr>
<tr>
<td>0174</td>
<td>CALL PPLIST (FEATS,6,6)</td>
</tr>
<tr>
<td>0175</td>
<td>CASEST = CAR(CAR(CAR(ROLES)))</td>
</tr>
<tr>
<td>0176</td>
<td>IF (CASEST =0,WHILE) CASEST = CAR(INF)</td>
</tr>
<tr>
<td>0177</td>
<td>S = NEW(FATS,CASEST,P,OUT,IFAR(CAR(ROLES))))</td>
</tr>
<tr>
<td>0178</td>
<td>IF (OUTP =0,WHILE) GOTO 106</td>
</tr>
<tr>
<td>0179</td>
<td>WRITE (6,111)</td>
</tr>
<tr>
<td>0180</td>
<td>FORMAT (AX, 'SUCCESSFUL TRANSITION IN SEMANTIC NETWORKS' )</td>
</tr>
<tr>
<td>0181</td>
<td>1 /AX,&quot;RESULTING TRIPLES (FEATURES * STATE * CASE)&quot;</td>
</tr>
<tr>
<td>0182</td>
<td>CALL PRLIST (OUTP,B,8)</td>
</tr>
<tr>
<td>0183</td>
<td>CALL PPLIST (OUTP,B,8)</td>
</tr>
<tr>
<td>0184</td>
<td>CASE = CAR(CAR(CAR(ROLES))))</td>
</tr>
<tr>
<td>0185</td>
<td>CALL PRLIST (CASE,6,6)</td>
</tr>
<tr>
<td>0186</td>
<td>OROLFS = SEARCH (CAR(CAR(NPFEAT)))</td>
</tr>
<tr>
<td>0187</td>
<td>IF (CAR(CAR(OROLFS)),FO,ICASE) GOTO 66</td>
</tr>
<tr>
<td>0189</td>
<td>OROLFS = CAR(OROLFS)</td>
</tr>
<tr>
<td>0190</td>
<td>IF (OROLFS =0,WHILE) GOTO 105</td>
</tr>
<tr>
<td>0192</td>
<td>GOTO 65</td>
</tr>
<tr>
<td>0193</td>
<td>OSMF = CAR(CAR(OROLFS))</td>
</tr>
<tr>
<td>0194</td>
<td>CALL PRLIST (OSMF,B,6)</td>
</tr>
<tr>
<td>0195</td>
<td>J = MATCH(OSMF,OSMF,SENTRE)</td>
</tr>
<tr>
<td>0196</td>
<td>IF (J =0,WHILE) GOTO 68</td>
</tr>
<tr>
<td>0197</td>
<td>WRITE (6,116)</td>
</tr>
<tr>
<td>0198</td>
<td>FORMAT (AX, 'SEM FEATURES MATCH SUCCESSFUL')</td>
</tr>
<tr>
<td>0199</td>
<td>CALL APPEND (CAR(CAR(I)),J,L)</td>
</tr>
<tr>
<td>0200</td>
<td>CALL APPEND (OUTP,CAR(I),OUTP)</td>
</tr>
<tr>
<td>0200</td>
<td>IN = IN + 1</td>
</tr>
<tr>
<td>0201</td>
<td>GOTO 67</td>
</tr>
<tr>
<td>0202</td>
<td>WRITE (6,117)</td>
</tr>
<tr>
<td>0203</td>
<td>FORMAT (AX, 'SEM FEATURES MATCH')</td>
</tr>
<tr>
<td>0204</td>
<td>T = CAR(I)</td>
</tr>
</tbody>
</table>

- 3.76. -
parser implementation

0203  IF (T,NF,0) GOTO 65
0205  IF (COR(1) NE,0) GOTO 10
0207  OUTP = COR(1)
0208  CALL BACK(1)
0209  IF (T,EF,0) GOTO 1087
0211  & CONTINUE
0212  WRITE(6,105)
0105  FORMAT (1X, '>>> ALL TESTS SUCCESSFUL, NEW CONFIGURATION:*')
0212  DO 59 IN = 1,1
0213  IF (OUTP=FO,0) GOTO 59
0215  NSYM = CAR(COR(COR(COR(OUTP))))
0216  ICASE = CAR(COR(COR(COR(OUTP))))
0218  CASEST = CAR(COR(COR(COR(OUTP))))
0219  OUTP = COR(OUTP)
0220  C(1) CHA NG I N SUBORDINATE CONFIGURATION
0222  59  CONF = COPY (SUSTRUC)
0224  FES = COR(COR(COR(COR(NEW))))
0226  IF (CONF(ORULE,0),NF,OBJEC) GOTO 193
0228  C (A) FOR OBJECTS
0230  TI = COR(COR(COR(FES)))
0232  C(1) SYNT FEAT
0234  IF (CAR(13) NE,0) CALL ERASE(CAR(13))
0236  CAR(13) = NSYM
0238  C(II) SEM FEAT
0240  IF (CAR(COR(13) NE,0) CALL ERASE(CAR(COR(13)))
0242  130  CASE(COR(13)) = NSYM
0244  C(III) CASE
0246  CASE(COR(COR(13))) = ICASE
0248  GOTO 194
0250  C (B) ANJUNCTS
0252  193  IF (OFUNC.EQ.FEATBAL) CAR (COR(COR(COR(FES))) = NSYM
0254  IF (OFUNC.EQ.SYNNET) CAR(COR(FES)) = FIN
0256  IF (STYP NE,0) CAR(COR(COR(COR(FES)))) = STYP
0258  C(2) CONSTRUCT PARTICLE SUPERSTRUCTURE
0260  194  CALL NEW (NSTATE)
0262  NSTRUC = COPY (NCONF)
0264  191  NSTATE = NSTATE
0266  C RANGE
0268  IF (F,EF,1) GOTO 261
0270  C FOR DIRECTION LEFT TO RIGHT
0272  200  CALL APPEND (NSTATE,CAR(COR(DCONF)),J)
0274  DCONF = COR(DCONF)
0276  CALL PUSH(DCONF,0)
0278  GOTO 297
0280  C FOR DIRECTION RIGHT TO LEFT
0282  201  CALL APPEND (NSTATE,CAR(DCONF)),J)
0284  DCONF = COR(DCONF)
0286  CALL PUSH(DCONF,0)
0288  C PUSH ON NSTATE,INVERSE
0290  CALL PUSH(NSTATE,INVERSE)
0292  C MERGE
0294  PFEL = 0
0296  WNR = CAR(PFIN)

- 3.77 -
parser implementation

0256 HYPO = CAR(CAR(CDR(NPOIN)))
0257 ISTRUCT = CDR(ISTRUC)
0258 NPOIN = NPOIN (ISTRUC, NPOIN, HYPO)
0259 I = CDR(NPOIN)
0260 K = CDR(CDR(CDR(I)))
0261 IF (CDR(I), EN, H) GOTO 190
0262 IF (CAR(CDR(I)), EN, H) GOTO 191
0265 I = CDR(I)
0266 GOTO 192
0267 191 CALL BACK(CDR(I))
0268 192 CALL APPEND (I, ONEW, J)
0269 I = 1
0270 IF (F, NE, 0) GOTO 52
0277 J = CDR(UNPK)
0273 53 IF (CDR(I), EN, H) GOTO 51
0275 IF (CAR(CDR(I)), EN, H) GOTO 52
0277 I = CDR(I)
0278 GOTO 53
0279 51 CALL APPEND (I, 0, 1)
0280 52 IF (CAR(CDR(I)), EQ, PREDIC) PFL = 1
0282 FET = CDR(CAR(CDR(NPOIN)))

C SYNTACTIC STATE
CAR(ISTRUC) = NEWS
C(3) CHANGES IN HEAD CONFIGURATION
0284 202 IF (ANews, NE, 0) CAR (CDR(FFITS)) = ANews
C(I) STATE IN CASE NETWORK
0286 203 I3 = CDR(CDR(FFITS)))
0287 IF (CASEST, NE, 0) CAR(CDR(CDR(FFITS))) = CASEST

C HEAD IS OBJECT
C(III) SYNTACTIC FEATURE COMPLEX
0289 204 IF (NSYN, EQ, 0) GOTO 205
0291 IF (CAR(I3), NE, 0) CALL ERASE(CAR(I3))
0293 CAR(I3) = NSYN
0294 205 IF (COMP(RULE, 2), NE, OBJEC) GOTO 206
C(IV) SEM FEATURE COMPLEX
0296 IF (NRES, EQ, 0) GOTO 196
0298 IF (CAR(CDR(I3)), NE, 0) CALL ERASE(CAR(CDR(I3)))
0300 CAR(CDR(I3)) = NRES
0301 GOTO 196

C HEAD IS ADJUNCT
0302 206 IF (CHAR, NE, 0) CAR(CDR(I3)) = CHAR
C VERBS

0306 196 IF (PFL, EQ, 0) GOTO 197
0308 J = I
0309 I = CDR(CAR(NSTATE))
0310 CALL APPEND (CDR(CAR(CDR(NSTATE))), I, L)
0311 L = CDR(J)
0312 CALL BACK (L)
0313 198 CAR(CAR(NSTATE)) = PREDIC
0314 197 IF (CAR, EN, H) CALL PRLIST(CAR(NSTATE), A, 6)
0316 58 CONTINUE
0317 RETURN

C END MESSAGES
0318 1001 IF (N0, EQ, 0) RETURN
0319 WRITE(E, 1011)

- 3.78 -
FORMAT (8x, "** WRONG HEAD OR NO TRANSITION IN SYNT NET*)

0320 IF (OJ.EQ.0) RETURN
0321 WRITE(6,1012)
0322 FORMAT (8x,"** SYNTACTIC FEATURES MATCH UNSUCCESSFUL")

0323 IF (OJ.EQ.0) RETURN
0324 WRITE(6,1013)
0325 FORMAT (8x,"** SEMANTIC FEATURES MATCH UNSUCCESSFUL")

0326 IF (OJ.EQ.0) RETURN
0327 WRITE(6,1014)
0328 FORMAT (8x,"** HEAD TAKES NO OBJECTS OR WRONG POSITION")

0329 IF (OJ.EQ.0) RETURN
0330 WRITE(6,1015)
0331 FORMAT (8x,"** NO TRANSITION IN SYNT NET")

0332 IF (OJ.EQ.0) RETURN
0333 WRITE(6,1016)
0334 FORMAT (8x,"** NO TRANSITION IN SYNT NET")

0335 IF (OJ.EQ.0) RETURN
0336 WRITE(6,1017)
0337 FORMAT (8x,"** SEMANTIC FEATURES MATCH UNSUCCESSFUL")

0338 END
parser implementation

NPOINT

parameters: STUC, WOR, HYPO

Operation:
This small auxiliary function is used to locate in a configuration (pointed at by STUC) 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 NP(OINT (ISTRUC, WOR, HYPO)
IMPLICIT INTEGER (A-N)
CALL NEW(PDS)
193 IF(CAR(ISTRUC),NE,WOR) GOTO 198
   IF(CAR(CDR(ISTRUC)),NE,Hypo) GOTO 196
   NPOINT = ISTRUC
   IF(POS,EQ,0) RETURN
   CALL PUPUP(I,PDS)
   GOTO 1
193 ISTRUC = CDR(ISTRUC)
   IF(CAR(CDR(ISTRUC)),E0,0) GOTO 192
   IF(CAR(CDR(ISTRUC)),E0,0) GOTO 192
   CALL PUPUP(ISTRUC,PDS)
   ISTRUC = CDR(ISTRUC)
   ISTRUC = CAR(ISTRUC)
   GOTO 193
192 CALL PUPUP(ISTRUC,PDS)
   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:

```fortran
INTEGER FUNCTION FRAMES (FEAT1, FEAT2, STYPE, SEMF)
IMPICIT INTEGER (A-N)
LOGICAL*1 AF
COMMON/COMP/LOCK, RULE, REFORP, AFTER, TRUE, FALSE, UNDET, FUNCTION,
* SYMF, FRAMES, OBJECT, UNMA, PREDIC
COMMON/COMP/COMP (39, 19)
COMMON/COMP/COMP (39, 19)
COMMON CAR(SYMP), COR(SYMP), AF (3002)
COMMON CAR(SYMP), COR(SYMP), AF (3002)
COMMON CAR(SYMP), COR(SYMP), AF (3002)
COMMON CAR(SYMP), COR(SYMP), AF (3002)
C GET CASE FRAMES
0099 IF FEAT1 = 0
0101 IFNAM = CAR (COR (FEAT2))
0103 IFNAM = CAR (COR (FEAT2))
0105 IF (IFNAM, EQ, 0, OR, JERNAM, EQ, 0) GOTO 8
0107 JROLES = SEARCH (JRNAM)
0109 JROLES = SEARCH (JRNAM)
0111 JROLES = SEARCH (JRNAM)
0113 JROLES = SEARCH (JRNAM)
0115 JROLES = SEARCH (JRNAM)
0117 IF (JROLES, EQ, 0, OR, JROLES, EQ, V) GOTO 8
0119 IF (JROLES, EQ, 0, OR, JROLES, EQ, V) GOTO 8
0121 IF (JROLES, EQ, 0, OR, JROLES, EQ, V) GOTO 8
0123 IF (JROLES, EQ, 0, OR, JROLES, EQ, V) GOTO 8
0125 IF (JROLES, EQ, 0, OR, JROLES, EQ, V) GOTO 8
0127 IF (JROLES, EQ, 0, OR, JROLES, EQ, V) GOTO 8
0129 IF CASE = CAR (COR (COMP (COR (FEAT2))))
0131 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0133 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0135 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0137 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0139 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0141 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0143 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0145 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0147 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0149 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0151 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0153 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0155 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0157 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0159 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0161 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0163 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0165 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0167 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0169 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0171 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0173 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0175 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0177 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0179 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0181 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0183 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0185 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0187 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0189 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0191 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0193 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0195 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0197 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0199 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0201 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0203 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0205 IF (CASE, EQ, 0, OR, ICARE) GOTO 3
0207 IF SEMF = CAR (COMP (CAR (JROLES)))
0209 WRITE (6, 11)
0211 FORMAT (44, "INVESTIGATE THE FOLLOWING SFM.FEATURES")
0213 CALL PHLIST (SEMF, R, 6)
```

- 3.81 -
parser implementation

C SEARCH FEATURES IF SLOT FILLER
C (A) QUALIFYING
0026  IF ((TYPE, EQ, MON)) GOTO 7
0028  C (B) MODIFYING
0029  IF (SEMFF, NE, ?) GOTO 6
0030  ICASE = CAR(COR(COR(COR(Feat1))))
0031  IF (CAR(COR(JROLES1)), EQ, ICASE) GOTO 5
0033  JROLES = COR(JROLES1)
0034  TF (JROLES, NE, ?) GOTO 4
0035  GOTO 14
0036  C COMPARE
0037  SEMFF = EXTRACT(CAR(COR(COR(JROLES1))))
0038  IF (FRAMES, MATCH (SEMFF, SEMFF, SEMFF)) GOTO 6
0039  CALL PLIST (SEMFF, a, b)
0040  TF (FRAMES, EQ, SEMFF) GOTO 7
0041  C (C) MODIFYING
0042  IF (TYPE, EQ, DUAL) RETURN
0043  IF (TYPE, EQ, UNDET) STYPE = DUAL
0044  IF (STYPF, EQ, ?) GOTO 6
0045  CALL PLIST (ISEMF, c, d)
0046  FRAMES = MATCH (SEMFF, ISEMF, SEMFF)
0047  IF (FRAMES, EQ, ?) GOTO 17
0048  TF (FRAMES, EQ, UNDET) STYPE = UNDET
0049  IF (FRAMES, EQ, I) FRAMES = IFR
0050  GOTO 12
0051  C ERROR
0052  WRITE (a, 9)
0053  FORMAT (ix, "MISSING FRAME")
0054  RETURN
0055  WRITE (a, 11)
0056  FORMAT (ix, "MISSING CASE IN FRAME")
0057  RETURN
0058  END
3.3. The computation of the structures

We present now three subroutines which extract the linguistic information structures 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

```
0001 SUBROUTINE FUN (CONF)
0002 IMPLICIT INTEGER (A-N)
0003 LOGICAL *1 AF
0004 COMMON/FIN/FIN,TR
0005 COMMON CAR(3000),AF(3000)
0006 IF(CONF,Eq.,F) RETURN
0007 CALL New(PDS)
0008 CALL INF(K)
0009 OUTFUN=FUNK
0010 NILFUN=CONF
0011 1 IF(FUN(CAR(CAR(CAR(INWOR))))
0012 J=CONF(CAR(CAR(CONF(INWOR)))))
0013 IF (CAR(J),EQ.,OR,CAR(J),EQ.,FIN)) GOTO 3
0014 IF (ELEM(FIN,CAR(J)),EQ.,F) GOTO 50
0015 J = CAR(CAR(J))
0016 IF (J,Eq.,OR,J,Eq.,FIN)) GOTO 4
0017 IF (ELEM(FIN,J),Eq.,F) GOTO 50
0018 J = CAR(OUTFUN)*CAR(INFUN)
0019 IF (J,Eq.,OR,J,Eq.,FIN)) GOTO 2
0020 IF (FIN,F) GOTO 2
0021 CALL INF(OUTWOR)
0022 CALL APEND(OUTWOR,OUTWOR)
0023 CALL APEND(OUTFUN,OUTWOR,13)
0024 IF (FIN,F) CALL APEND(OUTFUN)
0025 CALL APEND(OUTWOR,OUTFUN,13)
0026 CALL APEND(OUTWOR,OUTWOR,13)
0027 IF (FIN,F) GOTO 1
0028 CALL PUS(1,1,PDS)
0029 CALL PUS(1,1,PDS)
0030 GOTO 1
```

- 3.83 -
(ii) Case structures

CAS

parameters: CONF, a configuration

operation:
CAS computes the case structure and prints it on an output device.

code:

0001 SUBROUTINE CAS(CONF)
0002 IMPLICIT INTEGER(A-Z)
0003 LOGICAL*1 AF
0004 COMMON CASE(5999),COR(3999),AF(3999)
0005 COMMON CODE,MOD,DUAL,ADJU
0006 COMMON /COMP/ COMP(39,19)
0007 COMMON /CODE/ LOCK,RULE,BEFORE,AFTER,TRUE,FALSE,UNIQUE,FUNCT4,
0008 * SYPYET,FRAMET,OBJEC.INMA,PREDIC
0009 CAST = FRAME
0010 IF(CASE.EQ.4) RETURN
0011 CASE = CASE
0012 CASE = CASE
0013 CALL PUSH(CONF,POST)
0014 CALL PUSH(0,POST)
0015 IF(P.EQ.0) GOTO 99
0016 IF(P.EQ.0) GOTO 99
0017 IF(P.EQ.0) GOTO 99
0018 IF(CASE.EQ.4) RETURN
0019 IF(FLAG.EQ.0) GOTO 99
0020 IF(FLAG.EQ.0) GOTO 99
0021 IF(FLAG.EQ.0) GOTO 99
0022 IF(FLAG.EQ.0) GOTO 99
0023 IF(FLAG.EQ.0) GOTO 99
0024 IF(FLAG.EQ.0) GOTO 99
0025 IF(FLAG.EQ.0) GOTO 99
0026 IF(FLAG.EQ.0) GOTO 99
0027 IF(FLAG.EQ.0) GOTO 99
0028 IF(FLAG.EQ.0) GOTO 99
0029 IF(FLAG.EQ.0) GOTO 99
0030 IF(FLAG.EQ.0) GOTO 99
structuring

0028 12 FL=1
0029 1 PI=PI NHP(CPIW)
0030 1 IF PI=PI NHP(CPIW) AND (CAR(CPIW)=,EQ,1)) GOTO 1
0031 1 P=CAR(CPIW)
0032 17 P=PI NHP(CAR(CDR(P)))
0033 1 CALL GET(CAR(P),RULE,1R)
0034 1 IF (CONF(IP,2),NE,OBJEC) GOTO 6
0035 1 P=P=PI NHP(CAR(CDR(CDR(CAR(CDR(P))))))
0036 1 IF (P=PI NHP(CAR(CDR(CDR(CAR(CDR(P)))))))
0037 1 IF (FLG=1, 1, 1) GOTO 6
0038 1 CALL NEW(TX)
0039 1 CALL APPEND(CS,TX,CS)
0040 1 CAR(TX)=CA(R(P)
0041 1 CALL IF(I,1)
0042 1 CALL APPEND(TX,MY,TX)
0043 1 CAR(MY)=PRA
0044 1 CALL APPEND(MY,CA(R(P),MY)
0045 1 FLG=1
0046 1 PAN=PI NHP(CAR(P)))
0047 1 IF (P=PI NHP(CAR(CDR(CAR(CDR(P))))), EQ,1)) GOTO 18
0048 1 CALL PUSH(P,PSNR)
0049 1 CALL PUSH(P,PO)
0050 14 IF (POSSP,NE,0) GOTO 15
0051 1 GOTO 5
0052 6 IF (CONF(IP,2),NE,OBJEC) GOTO 14
0053 6 CALL PUSH(P,PSNR)
0054 6 CALL PUSH(P,PO)
0055 6 IF (POSSP,NE,0) GOTO 15
0056 6 GOTO 5
0057 14 IF (CONF(IP,2),NE,OBJEC) CALL PR LIST(CONF(IP,2),0,0)
0058 6 CALL PUSH(P,PSNR)
0059 6 PAN=PI NHP(CAR(P))
0060 15 P=PI NHP(CAR(CAR(CAR(P))))
0061 6 IF (P=PI NHP(CAR(CDR(CAR(CDR(P))))), EQ,1)) GOTO 16
0062 6 P=CAR(CAR(CAR(P))
0063 6 GOTO 17
0064 6 CALL PUSH(P,PSNR)
0065 6 IF (POSSP,NE,0) GOTO 15
0066 6 GOTO 5
0067 11 IF (CONF(IP,2),NE,OBJEC) GOTO 1
0068 1 CALL GET(CAR(P),CAR(CAR(CAR(P)))))
0069 1 VIEWP = CAR(CAR(CAR(CAR(CAR(CAR(CAR(CAR(CAR(CAR(P)))))))))))
0070 1 IF (FLG=1, 1, 1) GOTO 13
0071 1 CALL NEW(TX)
0072 1 CALL APPEND(CS,TX,CS)
0073 1 CAR(TX)=CA(R(P)
0074 19 CALL NEW(NX)
0075 1 CALL APPEND(TX,NX,TX)
0076 1 CAR(MY)=VIEWP
0077 1 CALL APPEND(NX,CAR(T),NX)
0078 1 FLG = 1
0079 1 GOTO 2
0080 90 CALL PR LIST(CASE,1,H)
0081 0 CALL FLATLI(CASE,1,1,1)
0082 0 RETURN
0083 0 END

- 385 -
(iii) Semantic structure

SEM

parameters: CONF, a configuration

operation:
SEM computes the semantic structure and prints it on an output device

code:

```
0001 SUBROUTINE SEM(CONF)
0002 IMPLICIT INTEGER(A-X)
0003 LOGICAL * 1, AF
0004 COMMON CAR(3000), CDR(3000), AF(3000)
0005 COMMON SEM/LIST, SEMSTR, PRED, ARG, FEAT, MOD, OBJEC, ADJU, FUNCTW
0006 COMMON CONF/CONF(30,10)
0007 COMMON ADD/RULE
0008 D NUM=0
0009 D PWNF=0
0010 D WRITE(6,101)
0011 D 101 FORMAT(1X,'CREATING TOP OF SEMANTIC STRUCTURE')
0012 CALL NEW(SEMA)
0013 C(SEMA)=SEMSTR
0014 SM=SEMA
0015 D WRITE(6,102)
0016 D 102 FORMAT(1X,'CREATING INITIAL TASK IMAGE')
0017 CALL PUSH(CONF,PDSO)
0018 CALL PUSH(P,PDSO)
0019 CALL PUSH(0,PDSO)
0020 CALL PUSH(2,PDSO)
0021 CALL PMLIST(PCO,9,6)
0022 1 D NUM=NUM+1
0023 D WRITE(6,146) NUM
0024 D 146 FORMAT(1H0,1H1,1H2,1H3)
0025 D WRITE(6,103)
0026 D 103 FORMAT(1H0,'.I. POPPING UP NEW TASK IMAGE')
0027 D WRITE(6,104)
0028 D 104 FORMAT(5X,'PRESENT POINT IN CONFIGURATION:')
0029 D CALL PMLIST(PCO,9,6)
0030 D CALL POPUP(PCO,PDSO)
```
structuring

0031 D WRITE(6,105)
0032 D 105 FORMAT(5X,"ATTACHMENT POINT IN SEMANTIC STRUCTURE!")
0033 D CALL PRLIST(PSE,9,6)
0034 D CALL POPUP(MQOX,PSDX)
0035 D WRITE(6,106)
0036 D 106 FORMAT(5X,"TOP OF NODE (FOR QUAL)!")
0037 D CALL PRLIST(MQOX,9,6)
0038 D CALL POPUP(MQPR,PSDP)
0039 D WRITE(6,107)
0040 D 107 FORMAT(5X,"PREDICATE NODE (FOR MOD)!")
0041 D CALL PRLIST(MQPR,9,6)
0042 D IF(PCO.EQ.0) GOTO 90
0043 D IF(PSE.EQ.0) GOTO 17
0044 D 18 IF(CDR(PSE),NE,0) PSE*CDR(PSE)
0045 D IF(CDR(PSE),NE,0) GOTO 18
0046 D WRITE(6,109)
0047 D 109 FORMAT(5X,"REASSIGNED ATTACHMENT POINT!")
0048 D CALL PRLIST(PSE,9,6)
0049 D 17 PFC=CDR(CAR(PFU))
0050 D IF(PCO.EQ.0) GOTO 19
0051 D WRITE(6,110)
0052 D 110 FORMAT(1HX,"II, EXECUTION OF TASK")
0053 D CALL PRLIST(CAR(PFU),30,6)
0054 D WRITE(6,111)
0055 D 111 FORMAT(1HX,"FUNCTION OF PRESENT WORD IS!")
0056 D IF(PCO.EQ.0) GOTO 19
0057 D WRITE(6,112)
0058 D 112 FORMAT(1HX,"II, EXECUTION OF TASK")
0059 D CALL PRLIST(CAR(OLIST),30,6)
0060 D P2A=PCO
0061 D P2FU=PFU
0062 D GOTO 16
0063 D P2=PFU
0064 D P2A=PFU
0065 D MQO=MQX
0066 D MQX=PR
0067 D WRITE(6,114)
0068 D 114 FORMAT(5X,"REASSIGNED ATTACHMENT POINT IN SEMANTIC STRUCTURE!")
0069 D WRITE(6,115)
0070 D 115 FORMAT(5X,"ATTACHMENT POINT IN SEMANTIC STRUCTURE!")
0071 D CALL PRLIST(PSE,9,6)
0072 D WRITE(6,116)
0073 D 116 FORMAT(5X,"TOP OF NODE (FOR QUAL)!")
0074 D CALL PRLIST(MQOX,9,6)
0075 D WRITE(6,117)
0076 D 117 FORMAT(5X,"PREDICATE NODE (FOR MOD)!")
0077 D CALL PRLIST(MQPR,9,6)
0078 D WRITE(6,120)
0079 D 120 FORMAT(1HX,"STARTING TO TRACE DEPENDENT WORDS")
0080 D GOTO 4
0081 D 19 IF(OLIST(PCO).EQ.0) GOTO 12
0082 D CALL PRLIST(CAR(OLIST),15,6)
0083 D WRITE(6,118)
0084 D 118 FORMAT(1HX,"PRESENT WORD: IS OBJECT-TYPE")
0085 D *1X,"STARTING TO TRACE DEPENDENT WORDS")
0086 D 4 PNH=CDR(PNH)
0087 D IF(PNH.EQ.0) GOTO 80
0088 D P2A=CAR(PNH)
0089 D WRITE(6,119)
0090 D 119 FORMAT(1HX,">> DEPENDENT WORD FOUND")
structuring

0091   25 P2FU=CDR(CAR(CDR(P2)))
0092   CALL GET(CAR(P2FU),RULE,IR)
0093      O CALL PRLIST(CAR(P2FU),18,6)
0094      O WRITE(6,121)
0095      O 121 FORMAT(1H+,4X,"FUNCTION IS")
0096      O IF(COMF(IR,2),NE,OBJEC) GOTO 7
0097      O CALL PRLIST(CAR(P2),11,6)
0098      O WRITE(6,122)
0099      O 122 FORMAT(1H+,4X,"WORD: IS OF OBJECT-TYPE")
0100      O *5*, "STARTING TO CREATE NEW OBJECT NODE"
0101      O CALL NEW(NPL)
0102      O OX=CAR(OLIST)
0103      O OLST=CDR(OLIST)
0104      O CAR(NPL)=OX
0105      O CALL APPEND(SM,NPL,SM)
0106      O NEW(PR)
0107      O CALL APPEND(NPL,PR,NPL)
0108      O CALL GET(CAR(P2),CAR(CAR(CDR(P2))),INF)
0109      O IMPR=CDR(INF)
0110      O CALL APPEND(PR,CAR(CDR(CDR(INF))))
0111      O CALL APPEND(PR,CAR(INPR),PR)
0112      O IF(CAR(CAR(INPR)),NE,P) CALL APPEND(PR,CAR(CDR(INPR)),PR)
0113      O FEAT=CDR(CDR(CDR(P2FU)))
0114      O IF(COMF(IR,2),NE,OBJEC) FEAT=CDR(FEAIN)
0115      O CALL FEAT(CAR(FEAIN),FEAOUT)
0116      O IF(FEAOUT,EQ,0) GOTO 20
0117      O CALL NEW(FE)
0118      O CAR(FE)=FEAT
0119      O CALL APPEND(NPL,FE,NPL)
0120      O CALL APPEND(FE,FEAOUT,FE)
0121      O WRITE(6,123)
0122      O 123 FORMAT(1H+,4X,"OBJECT NODA COMPLETED AND ATTACHED TO *
0123      O *SEMANTIC STRUCTURE")
0124      O CALL PRLIST(CAR(SM),7,6)
0125      O 20 P2CA=CDR(CDR(CDR(CDR(CDR(P2FU)))))
0126      O IF((COMF(IR,2),NE,OBJEC),(CAR(P2CA),EQ,0)) GOTO 2
0127      O CALL PRLIST(0X,27,6)
0128      O WRITE(6,124)
0129      O 124 FORMAT(1H+,4X,"NOW ATTACHING OBJECT TO ARGUMENTS")
0130      O IF(CAR(CAR(PSE)),EQ,ARG) GOTO 5
0131      O CALL NEW(ARG)
0132      O CAR(ARG)=ARG
0133      O CALL APPEND(PSE,AR,PSE)
0134      O CALL NEW(CA)
0135      O CAR(CA)=P2CA
0136      O CALL APPEND(ARG,CA,AR)
0137      O CALL PRLIST(CA,OX,CA)
0138      O CALL PRLIST(CAR(PSE),7,6)
0139      O P2NW=CDR(CDR(P2))
0140      O IF((P2NW,EQ,0),(CAR(P2NW),EQ,0)) GOTO 29
0141      O WRITE(6,125)
0142      O 125 FORMAT(1H+,17X,"HAS DEPENDENT WORDS = PUSH NEW TASK IMAGE")
0143      O CALL PUSH(NPL,POSSP)
0144      O CALL PUSH(P2,POSSP)
0145      O CALL PUSH(IN,POSSP)
0146      O CALL PUSH(PR,POSSP)
0147      O GOTO 27

- 3.88 -
CONTINUE

D WRITE(6,124)

D 126 FORMAT(1H4,17X,'HAS NO DEPENDENT WORDS')

27 IF(PDSP2,NE,0) GOTO 24

GOTO 4

CALL GET((CAR(P2FU),RULE,IR)

IF((CONF(IR,2),NE,ADJU) GOTO 8

CALL PRLIST((CAR(P)),11,6)

CALL PRINT((DIR,16,6)

D CALL PRLIST((DIR,16,6)

D 128 FORMAT(1H4,4X,'WORD')

D 129 FORMAT(1H4,6X,'SUBTYPE')

D 129 FORMAT(1H4,6X,'SUBTYPE')

D CONTINUE

D WRITE(6,124)

D 126 FORMAT(1H4,17X,'HAS NO DEPENDENT WORDS')

27 IF(PDSP2,NE,0) GOTO 24

GOTO 4

CALL GET((CAR(P2FU),RULE,IR)

IF((CONF(IR,2),NE,ADJU) GOTO 8

CALL PRLIST((CAR(P)),11,6)

CALL PRINT((DIR,16,6)

D CALL PRLIST((DIR,16,6)

D 128 FORMAT(1H4,4X,'WORD')

D 129 FORMAT(1H4,6X,'SUBTYPE')

D 129 FORMAT(1H4,6X,'SUBTYPE')
0206  CALL NEW(NPL)
0207  CAR(NPL) = OX
0208  CALL APPEND(PSE,NPL,PSE)
0209  CALL NEW(PR)
0210  CAR(PR) = PR
0211  CALL APPEND(NPL,PR,NPL)
0212  CALL GET(CAR(PCD),CAR(CAR(CDR(PCD))),INF)
0213     IMPR = CDR(INF)
0214  CALL APPEND(PR,CAR(CDR(INF)),PR)
0215  CALL APPEND(PR,CAR(IMPR),PR)
0216  IF(CAR(COR(IMPR)),NE,PR) CALL APPEND(PR,CAR(COR(IMPR)),PR)
0217  IF(IOX,EO,MOD) GOTO 2
0218  D   CALL PRLIST(MQOX,PO,6)
0219  D   WRITE(6,139)
0220  D   139 FORMAT(1H", "NOW ATTACHING TO: TO ARGUMENTS OF QUALIFIER")
0221  CALL NEW(A)
0222  D   CAR(A) = ARG
0223  CALL APPEND(NPL,AR,NPL)
0224  CALL NEW(CA)
0225  CAR(CA) = CAR(COR(COR(IMPR)))
0226  CALL APPEND(CA,CA,AR)
0227  CALL APPEND(CA,MQOX,CA)
0228  D   CALL PRLIST(CAR(NPL)),1,6
0229  D   CALL PRLIST(0X,3,6)
0230  D   WRITE(6,141)
0231  D   141 FORMAT(1H", "IM", TX,"NODE COMPLETED AND ATTACHED")
0232  D   CALL PRLIST(CAR(PSE),1,6)
0233  GOTO 2
0234  D   15 CALL PRLIST(CAR(PFU),39,6)
0235  D   WRITE(6,142)
0236  D   142 FORMAT(1H", "S ERROR S --CANNOT IDENTIFY FUNCTION")
0237  D   CALL PRLIST(CAR(PCD),39,6)
0238  D   WRITE(6,143)
0239  D   143 FORMAT(1H", "99X,"OF WORD")/
0240  D   +12X,"FOR INCORRECT INPUT FROM POPUP")
0241  GOTO 4
0242  D   80 CONTINUE
0243  D   WRITE(6,144)
0244  D   144 FORMAT(1X,"NO (MORE) WORDS DEPENDENT FROM PRESENT WORD"/
0245  D   +10X,"III, SEMANTIC STRUCTURE AT PRESENT STAGE")
0246  D   81 SMA=SEMA
0247  D   IF(SMA,EE,Q9) GOTO 82
0248  D   CALL PRLIST(CAR(SMA),7,6)
0249  D   GOTO 81
0250  82 GOTO 1
0251  99 WRITE(6,145)
0252  145 FORMAT(1H", "SEMANTIC STRUCTURE COMPLETED NOW"/
0253  10X,"FINAL OUTPUT")/
0254  D   SMA=SEMA
0255  91 SMA=COR(SMA)
0256  D   IF(SMA,EE,Q9) RETURN
0257  CALL PRLIST(CAR(SMA),7,6)
0258  CALL PLOT(CAR(SMA),1,1,1)
0259  GOTO 91
0260  END

- 390,-