Peter Bøgh Andersen:

FANGORN - A LANGUAGE FOR GENERATING COHERENT TEXTS

1. INTRODUCTION AND GENERAL IDEAS

Fangorn is a system that reads descriptions of texts and generates samples of the texts described. It can be used for checking the empirical adequacy of text descriptions: if the output deviates (in some sense) from the corpus intended to be covered by the description, then the description is empirically inadequate.

Since many texts relate narratives about humans acting in purposeful, although conflicting, ways, Fangorn must contain facilities for describing problem-solving algorithms and it can be used for experiments in that area. However, the emphasis is not on efficiency but on simplicity, and it is strongly oriented towards producing readable texts as output. In these respects it deviates from systems such as TALE-SPIN (Meehan(77)).

Fangorn is heavily influenced by SIMULA and to a lesser extent by LISP. It is being programmed in SIMULA, but I comtemplate reprogrammingit in a less expensive language when it is debugged.

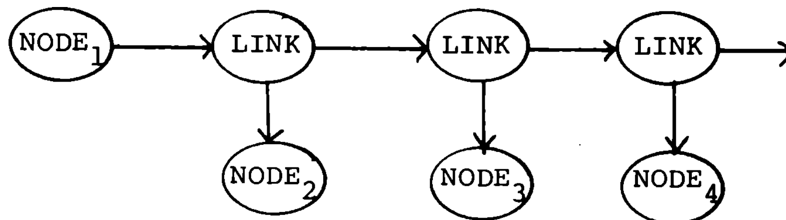
A Fangorn program is written in F-expressions (akin to LISP S-expressions). They are translated into a connected labelled network with a least upper bound. The network may contain cycles. It consists of NODES linked together with LINKs. Every NODE has certain attributes (variables) and in addition contains a block of actions that are executed when the node is activated. The attributes of a node are called its _structure_ and the actions are called its _process_. Objects containing a structure and a process are called _aggregates_. Every node in Fangorn is an aggregate, and every operation Fangorn can perform is a process, that is: it is associated with some node.

Every node has at least two attributes: a variable of type text, _name_, and a pointer variable, _suc_. LINKs have no name; instead they contain a pointer variable, _val_, pointing to a NODE.

FANGORN - A language for generating coherent texts
Peter Bøgh Andersen
Proceedings of NODALIDA 1979, pages 15-29

The basic network of FANGORN looks like this:

Fig.1



$NODE_1$ is <u>a mother of</u> nodes $NODE_2$, $NODE_3$, $NODE_4$. The latter are <u>daughters of</u> $NODE_1$. Two nodes that are daughters of the same node are said to be <u>sisters with respect to that node</u>.

Note that a node may have several mothers, and that two nodes may be sisters with respect to one node, but not with respect to another one.

Fig.1 may be drawn in a slightly simplified way as fig.2 or fig.3:

Fig.2

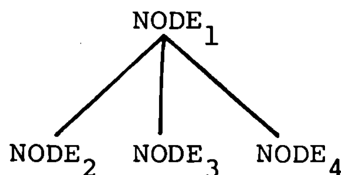
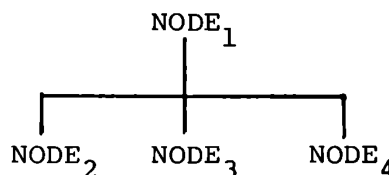


Fig.3



A major principle behind Fangorn is that a network must be able to reproduce itself, if it is to bear a likeness to natural language. We know that a natural language is thus structured that children, when exposed to it, learn it in an amazingly uniform way. We know too that no simple copying operation is involved (copying the contents of the adult brain into the child's brain or the like). Instead, the child's language is built up, stage by stage, and at each stage it is capable of functioning as a language. A grammar does not only produce sentences: it also reproduces itself at the very same time as it produces sentences,

and largely by means of the very same mechanisms it uses to produce those sentences.

For a "programming" language, this has the following consequence: the products it produces must be of the same kind as the program itself, and the operations by means of which it creates its output must be sufficient to create an output that is functionally equivalent to itself.

Of course, any programming language could be fixed to meet these requirements: if it contained a procedure run(file) that would compile and execute the program, written on file, then a program could write another program on file, possibly a copy of itself; run(file) would then compile and execute the program written on file. The difference between Fangorn and our hypothetical language is that the products of Fangorn are "programs" that may be executed without further ado, whereas our hypothetical program produces descriptions of programs that must be translated by very complicated processes before they can be executed.

Suppose that a parent network produced an offspring network in Fangorn: then the parent would be a very skillful educator of the infant, because the infant is structured as any other object that the parent can create and manipulate - the parent may use the same techniques it would use in any other situation when educating the child.

On the other hand, our hypothetical program would be a very poor educator: presumably, the child must be killed (the program must be terminated) and a new version written and compiled (born!) if changes are to be made. We just dont do such things nowadays!

If follows that a Fangorn program may change itself - it can educate itself. And this is obviously a desirable ability: because in many novels the protagonist changes during the narrative: for exampl his problem-solving algorithms may change as a result of successes, failures, new insights, or what have you.

These are the principial reasons why every bit of a Fangorn program is an energetic aggregate, ready to act when requested.

It has certain drawbacks, however. It makes recursive programming exetremely expensive, because every time a "procedure" is called, a copy of the <u>whole</u> procedure must be created. Copying the local variables of the procedure does not suffice, since the procedure may change its body during execution.

Fangorn is a forest in <u>The Lord of the Rings</u> by Tolkien, and since the program treats its network as a collection of trees, I thought that "Fangorn" fitted very well, the novel being one of my favourite books. But there is a little more to it than that: Fangorn is a very peculiar forest, consisting of trees, some of which are alive and move and act, and some of which are asleep and hard to wake. I have always been fascinated by Fangorn, because it contradicts the common idea of the world as consisting of two separate phenomena: things that are dead and passive, and beings that are alive and active. This conception simply does not fit language: a text, for example, is a thing: but it is also a process, influencing the reader in complicated ways, and leading him to conclusions that he may be most unwilling to draw.

In many programming languages, the passive-active dichotomy emerges as the rigid data-statement division, which may be a useful distinction in some areas, but is extremely cumbersome when natural language is concerned. I have sometimes wondered whether this stubborn insistence on the passive-active dichotomy might not be due to an underlying powerful ideology that classifies everything as either being passive, subordinate and willing to undergo manipulation, <u>or</u> as being capable of and entitled to doing the manipulation, with nothing in between. But such a division does not accord with the facts, even if it accords with the wishful thinking of the present potentates.

However that may be, in Fangorn I have tried to obliterate the distinction as far as possible, insisting that every action is performed by some entity that can itself be acted upon, and conversely, that every entity has at least a rudimentary action potential.

2. FLOW OF CONTROL

When a node is activated, it executes its process and then activates one of its daughters defined by its structure. Most nodes conform to the following pattern: they have at most 3 possible outcomes, success, failure and dont know, which correspond to its 3 rightmost daughters in that order. For example, the node BELIEVED may have 3 results: true, false or dont know. A person may believe a sentence, he may believe its negation or he may just dont know. If he believes it, the last-but-two sister is activated, if he disbelieves the last-but-one sister is activated, and if he dont know the last sister is activated.

Some nodes may have only one outcome, succes. For example, the node SET has 3 daughters; it assigns the daughters of its second daughter to its first daughter, and then activates its third daughter, thereby corresponding to the assignment statement.

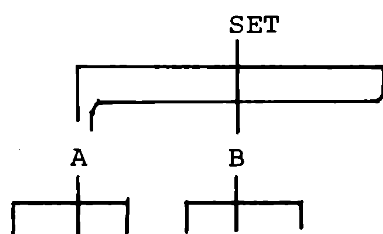
A node may have operands; they are always the youngest daughters. SET has two operands, its first two daughters.

Observe, that the format

(operands) + succes + (failure + (dontknow))

does not prevent a node from playing more than one role at a time. I fig.4, A plays the role of operand and at the same time functions as the succes-node:

Fig.4



The daughters of A are replaced by the daughters of B, and lastly A is activated.

3. MATCHING AND ASSIMILATING NETWORKS

Let A be a node. The network consisting of all nodes accessible from A via mother-daughter relations is called the network defined by A (or dominated by A). Instead of the phrase "the network dominated by A" I will sometimes write just "A" when no confusion results.

The network defined by A is said to match the network defined by B iff there is an isomorphism L from A into a subset of B, preserving

1. names
2. mother-daughter relations
3. sister relations

and such that

4. L(A) = B

Thus, node 1 matches node 7 but not node 18:

Fig.5

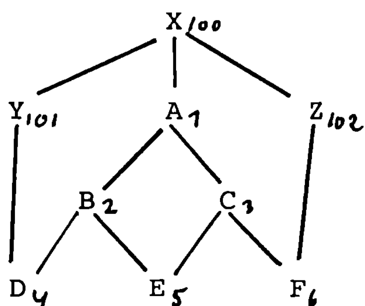


Fig.6

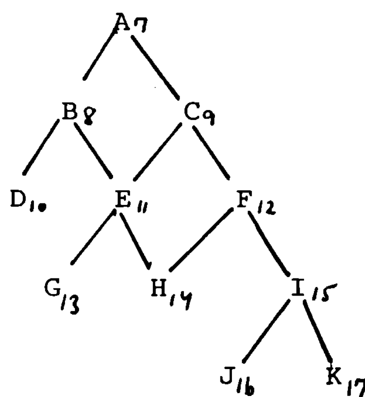
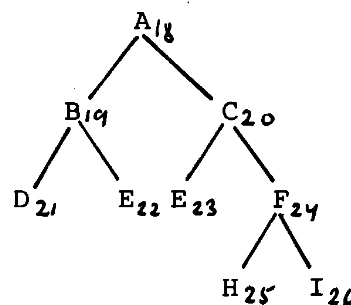


Fig.7



because there is an isomorphism L from 1 into 7,

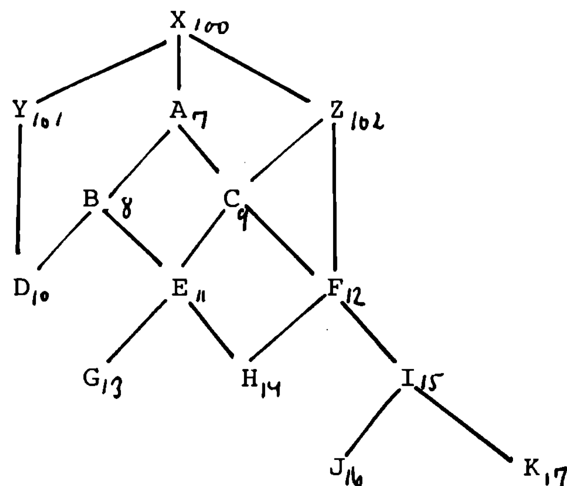
Fig.8

$$L = \{(1,7),(2,8),(3,9),(4,10),(5,11),(6,12)\}$$

If A matches B, then B is said to be an instance of A

If L is an isomorphism and S is any node, then ASSIM(S,L) is the network dominated by S, except that each node A in the domain of L is replaced by L(A). For example, if we assimilate node 1 into node 100 using the isomorphism L i fig. 8, then we get:

Fig.9



Assimilation, using the isomorphism produced by matching, is used in several different ways in Fangorn. Limitations of space prevent me from describing the processes in any detail, but I will give a rough outline. The relevant processes are:

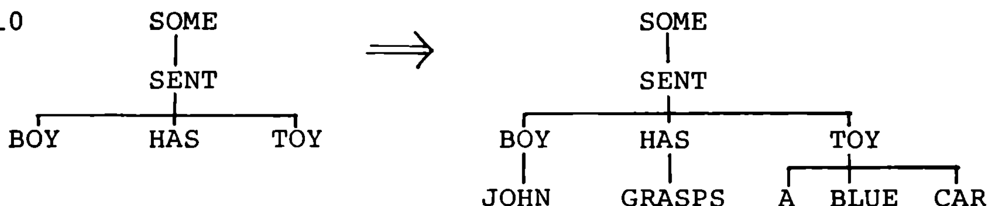
1. instantiation/ binding of variables (node: INSTANCE)
2. transformations in two varieties (node: REORDER)
3. expansions in two varieties (node: GROW and EXPAND)
4. anaphors (not implememted yet)

3.1 INSTANTIATION

In Fangorn it is possible to state conditions. There are four types of conditions, SOME, ALL, EXCEPT and NONE. A condition has exactly one daughter called its proposition. To give an example, SOME requires that at least one instance of its proposition be true at the "time" of the condition.

To instantiate a condition means to replace the proposition by certain sets of instances of the proposition or its negation. Example:

Fig.10



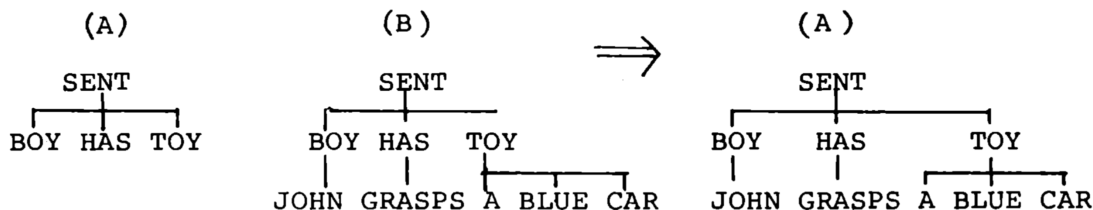
BOY, HAS and TOY may function as variables, to which the values JOHN, GRASPS and A BLUE CAR are assigned.

3.2. EXPANSIONS

We have two expansion atoms, EXPAND and GROW.

GROW is a generalisation of Chomsky´s rewrite rules; its process searches a list for a network B that is an instance of another network A. If A matches B, then B is assimilated into A´s context:

Fig.11



EXPAND is like GROW in that it searches a list in order to find matching nodes; but in this case B must be a daughter of a node X on the list, and B must match A and not conversely. If B is found, then A is assimilated into X, and X is inserted as the left sister of A. Example:

Fig.12

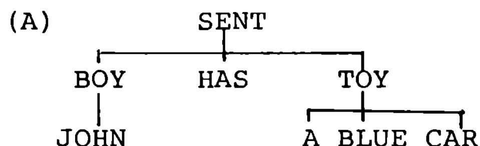
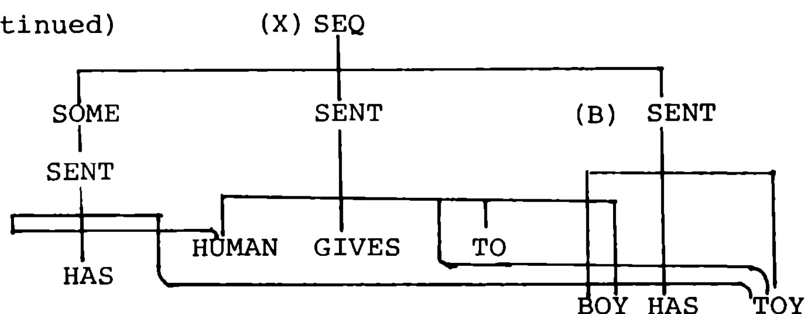


Fig.12 (continued)



Before assimilation the rule reads: if someone has a toy, and he gives it to a boy, then the boy has the toy. After assimilation, the rule reads: if someone has a blue car, and he gives it to John, then John has a blue car.

3.3. TRANSFORMATIONS

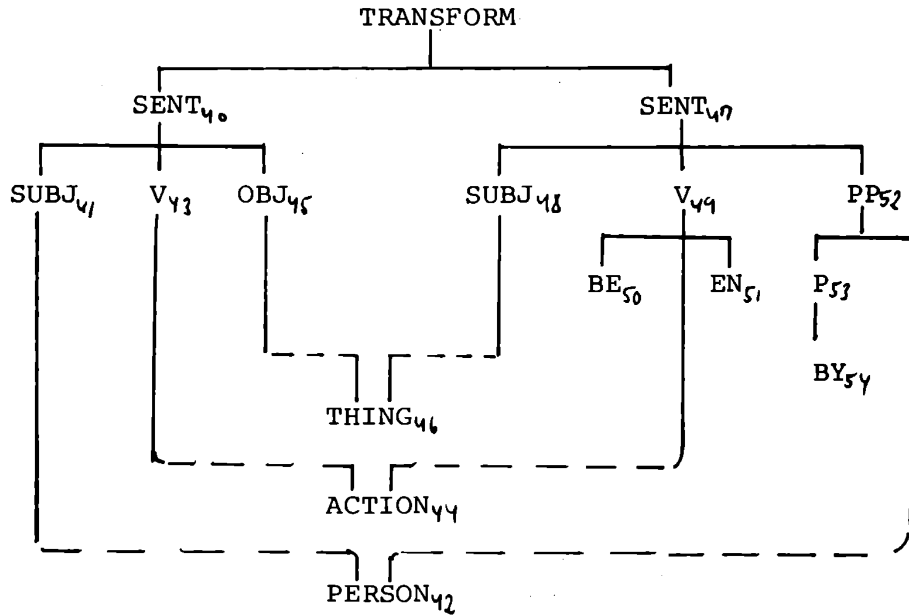
Both varieties, TRANSFORM and MOVE, consist of a structural description part (SD) and a structural change part (SC). A transformation is applicable to any network that its SD matches (actually, we allow two kinds of variables in the SD, the one being the X-variable of transformational grammar).

3.3.1 TRANSFORM

Let a SD match network A, producing isomorphism L. Then A is assimilated into SC using L. And this modified SC is equal to the result of the transformation.

Example: the passive transformation could be formulated:

Fig.13



$SENT_{40}$ is the structural description, and $SENT_{47}$ is the structural change. If fig.13 is applied upon fig.14 we get fig.15 :

Fig. 14

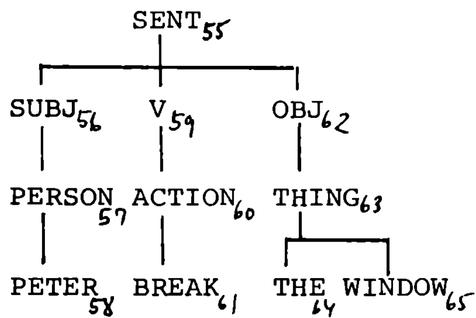
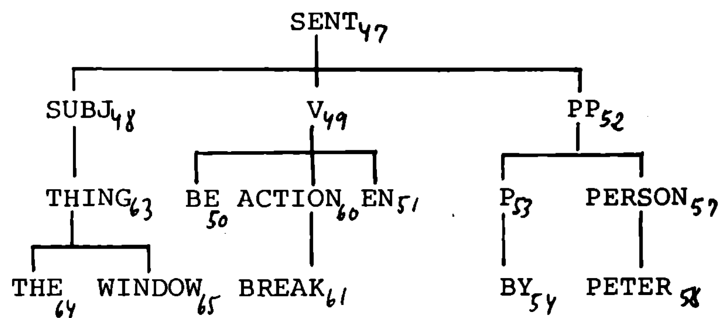


Fig. 15



40 matches 55, and assimilating 55 into 47 gives fig. 15.

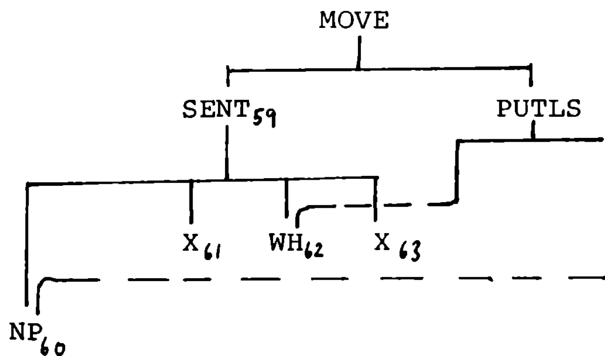
Not all transformations can be represented in this way, and therefore another format is supplied, called MOVE. The SD part is similar to the SD above, but the SC is different. It consists of orders as

ADD (as) L(eft)S(ister)
PUT (as) R(ight)D(aughter)

The first daughter of an order defines the object to be moved, and its second daughter defines the destination.

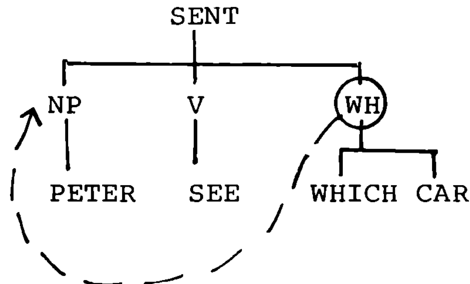
WH-movement could be represented thus:

Fig.16



The node defined by 62 is detached and attached as a left sister of the node defined by 60, eg:

Fig. 17



Expansions and transformations work on plans as well as on sentences: production of sentences and of plans are seen as essentially similar processes, both involving a grammar containing expansion rules and transformational rules, cf. BØgh Andersen (73, 77 and 78). Thus, sentence production is seen as a special kind of goal-directed behavior, that is: work.

4. A SIMPLE FANGORN PROGRAM

A FANGORN program is a connected network with an upper bound, and the input language is simply a description of this network using parentheses and anaphors. Fig.18 shows a simple FANGORN program, and fig.19 shows how it is described in the input language.

The program contains a list of expansion rules (RSET) and an algorithm for using these rules (the subnetwork dominated by DOWN). The rules are applied on the network dominated by GOAL. Most of the atoms in the algorithm refer to an implicit pointer C, that is: RIGHT moves C to its right sister, ISLASTSIS checks whether C is the last sister, etc.

The algorithm expands the SENT-node under GOAL into a sentence. Initially, the pointer C points to GOAL. DOWN moves it down to SENT, and GROW tries to expand the value of C, that is: SENT. If it succeeds the pointer is moved down to its first daughter, else we check whether C is the last sister. If not, then C is moved to the right and the new pointer value is expanded. If C is the last sister, then we check whether C points to GOAL (ISTOPGOAL). If not, C is set to its mother(UP) and we check whether C is now the last sister. If C points to the top goal we have finished, and the sentence is written on a file called OUTDATA. Then the pointer is moved to the ACTOR node (TOP) and the whole actor is written on a file named PAPER. Then the algorithm stops.

The boxed portion of fig.18 shows a sentence generated by the program.

OUTDATA and PAPER are "channels" connecting the FANGORN program to the file system. OUTDATA and PAPER belong to different types of channels: when a network is written on OUTDATA only its leaves (or terminal nodes) are printed, so OUTDATA is oriented towards accepting natural language texts. When a network is written on PAPER it is translated into the input language, so PAPER can be used for storing and retrieving parts of the FANGORN program.

Fig.18 is very simple and does not generate stories or coherent texts, but facilities for these tasks are present in the program. In other programs, EPIC will have daughters representing the sequence of actions performed by the ACTORs. CAST may contain more than one actor, acting in a pseudo-parallel way. The algorithm in fig.18 may be replaced by algorithms for building and executing plans; in that case, RSET contains means-end rules, and CONT dominates

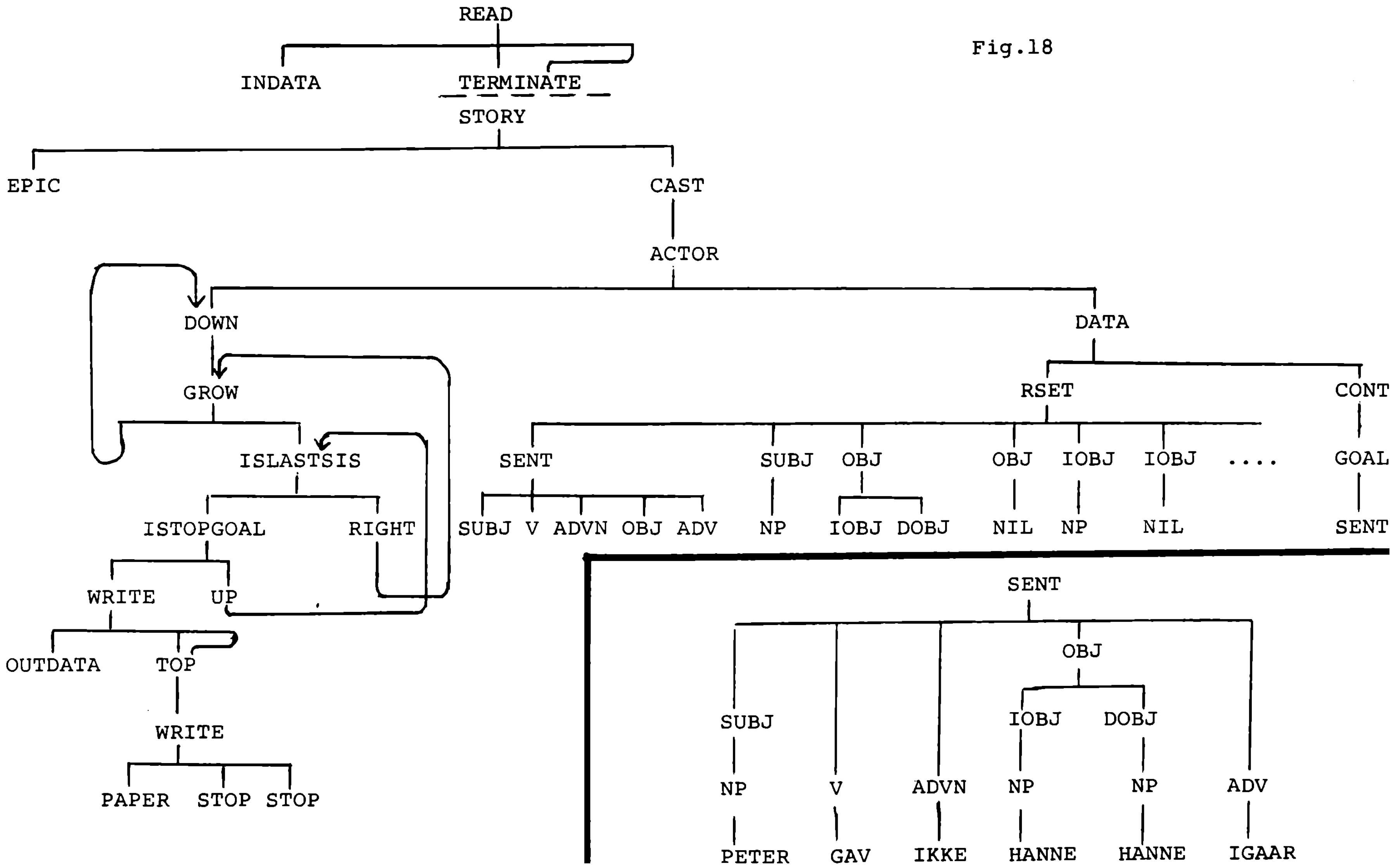


Fig.18

Fig.19

```
(ACT ACTOR ALL BACKTRACK BELIEVED CAST CURRENT DEEP
 DELETE DOWN EMBRYO EXCEPT FIRSTLEAF GROW HASMOTHER
 INSTANCE ISCOND ISFIRSTSIS ISLASTLEAF ISLASTSIS ISLEAF
 ISPROP ISSEQ ISTOPGOAL LEFT NEGATE NEXTACTOR NONE READ
 REORDER RIGHT SATISFY SET SOME STACK STOP STORY TERMINATE
 TEST TIMEOUT TOP TOPGOAL TRACK TRUE UNSTACK UP WRITE)

(STORY
  EPIC
  (CAST
    (ACTOR
      (0! DOWN
        (1! GROW
          0? DOWN
          (2! ISLASTSIS
            (ISTOPGOAL
              (WRITE
                OUTDATA
                (3! TOP (WRITE PAPER STOP STOP))
                3? TOP
              )
              (UP 2? ISLASTSIS)
            )
            (RIGHT 1? GROW)
      ) ) )
      (DATA
        (RSET
          (SENT SUBJ V ADVN OBJ ADV)
          (SUBJ NP)
          (OBJ IOBJ DOBJ)
          (OBJ NIL)
          (IOBJ NP)
          (IOBJ NIL)
          (DOBJ NP)
          (NP PETER)
          (NP HANNE)
          (NP HUNDEN)
          (NP KATTEN)
          (V GAV)
          (V BRAGTE)
          (ADVN IKKE)
          (ADVN NIL)
          (ADV IGAAR)
          (ADV SIDSTE JUL)
          (ADV NIL)
        )
        (CONT (GOAL SENT))
) ) ) )
```

a network representing the beliefs and goals of the actor.

The implicit pointer C may be moved into the "algorithmic" part of the actor, and the actor may thus change himself.

Also, an actor may contain rules for generating new actors; when they are fully developed, they may be raised from the goal part into the CAST, and start interacting with the older actors.

It is possible to simulate the ontogenetic or phylogenetic development of language by means of a meta grammar that contains an object grammar in its goal part. The meta grammar generates part of the object grammar, activates the object grammar, thereby causing it to produce sentences; control is returned to the meta grammar,which enlarges or transforms the object grammar, again the latter is activated, etc., etc. The output will consist of sentences from encreasingly more complex grammars; the aim is to write a meta grammar that produces a sequence of object grammars whose sentences mirrors the development of linguistic skill in children.

At the time of writing, the FANGORN system is implemented but not debugged. It contains 47 different atoms, but I plan to add approximately 10 new atoms, so the final number will be 55-60.

REFERENCES

Peter Bøgh Andersen: Handlinger og symboler. Elementer af handlingens syntaks (Akademisk Forlag, 1973)

" : Sproget på arbejde (GMT, 1977)

" : "The syntax of texts and the syntax of actions" (in: Pragmalinguistics, ed. J.L.Mey, Mouton,1979)

J. R. Meehan : "TALE-SPIN, an Interactive Program that Writes Stories" (in: 5th Int. Joint Conf. on Art. Int., 1977, p.91 - 98)

Earl D. Sacerdoti : "The nonlinear nature of plans" (in: 4th Int. Joint Conf. on Art. Int., 1975, p.206 - 214)

" : The structure for plans and behavior (Elsevier Computer Science Library, Elsevir, 1977)

John F. Sowa : "Conceptual Graphs for a Database Interface" (IBM Journal of Research and Development, July,1976)