ANTWERP PAPERS IN LINGUISTICS

1975, nr. 3

COMPLETION GRAMMARS AND THEIR APPLICATIONS

Luc Steels

UNIVERSITEIT ANTWERPEN

Universitaire Instelling Antwerpen

Departementen Ger. en Rom.. Afdeling Linguïstiek

Universiteitsplein 1 - 2610 Wilrijk - TEL 031/28.25.28

ABSTRACT

Three new types of grammars: open, closed and complex completion grammars, are formally defined and their relative parsing systems are discussed.
Also it is shown how these systems together with interpretation mechanisms make up complete language understanding systems. The applicability is illustrated by computer programmed experiments in natural and artificial language processing.

The basic novelties are a new approach towards the internal order of the elements in a language expression, the introduction of structures, distinct from constituent structure trees, for representing the information necessary for semantic interpretation and a strong procedural attitude towards language theory, as well on a syntactic as a semantic level.

Completion grammars in general can serve as a model for functional or relational grammars in that the emphasis on order, which is basic to the concept of constituent structure grammars, is replaced by emphasis on internal relations due to semantic properties. As a result the currently widely accepted distinction between deep and surface structures becomes unnecessary. Indeed with the grammars defined it is possible to map the language input directly into structures which contain all the information for a semantic interpretation.

CONTENTS

Introduction

This paper is devoted to the study of systems designed for the task of language processing. The main theme is the investigation of one part of such systems, namely the one by which analysis is being done. Analysis is the task of mapping the language input into a formal structure upon which interpretation takes place.

In the following sections we will introduce three types of grammars designed for the purpose of analysis: open completion grammar (§ 1) , closed completion grammars (§ 2) and complex completion grammars (§ 3) . We will also define parsing systems for the three types of grammars and interpretation mechanisms.

This paper is a statement on research in progress. Therefore we do not present fully worked out discussions, but only give an indication of the way in which research proceeds. The emphasis is on the definition of a fundamental framework, applying it to the data is another matter.
This does not mean however that we do not give any explicit information at all. The systems will be formally defined, the algorithms are all programmed and we will present concrete experiments in natural and artificial language processing. In particular we programmed and implemented experimental versions of language processing systems based on the 3 types of grammars and do experiments with as source languages the Propositional Calculus (in infix, prefix and postfix notation) and (subsets of) a natural language.

We thank the members of the reading committee especially prof. De Schutter, prof. Goossens, prof. Tasmowski because they accepted the paper for publication and made some very helpful remarks to improve the text. We also thank prof. Rozenberg D. Vermeir and H. Daman for discussing the matter and providing new insights into it. Of course responsibility for all remaining errors and deficiencies remains by the author.

0. FUNDAMENTALS OF SEMANTIC INTERPRETATION

Basically we assume that a predicate is a procedure or function name. The meaning of a predicate is equal to a procedural definition of its corresponding function and the interpretation of a predicate is equal to the execution of the procedure.

Consider e.g. 'SUM'. The procedure 'sum', familiar from simple arithmetics, takes two numbers as input and returns another appropriate number (sometimes called the 'value' of the procedure) as output. Understanding what'sum'means is considered to be the same as knowing what the procedure is and being able to execute the procedure on a given input.

A procedure calls for certain arguments (also called operands or cases) as input. These arguments are either resulting as values from other procedures (then the arguments will be called hidden) either directly present in the language expression (then the arguments will be called occurred).

Consider 'the sum of 1 and 1'. The procedure here is again 'sum', arguments are '1' and '1'. Similarly consider '2 times the sum of 6 and 2'. 'Sum' takes now '6' and '2' as arguments, the result is 8. This result together with '2' is input to the procedure 'times'. Note that '8' is an hidden argument, '6', '2' and '2' are occurred ones.

In a text one does of course not meet expressions as 'a,b and c are input to the procedure A and f is output', this would be a tedious way of communicating Instead we find simply 'A a b c ' or ' a A b c ', i.e. input arguments and procedures are written after each other and finding the exact input and outputrelations is left to the person trying to understand. So, a language expression will be considered as a series of procedure names and arguments.

The task of understanding consists in:

(i) finding out how the procedures denoted by these procedure names are interrelated with the arguments (this phase is called analysis) and

(ii) executing the procedures (this phase is called interpretation).

The problem of analysis or in other words the problem of extracting from a natural or artificial language input the corresponding semantic structure, will be solved by the definition of grammars (in particular completion grammars) and parsers, being systems computing the structures assigned by the grammar to an arbitrary combination in the language. The solution to the problem of interpretation involves a definition of all the procedures for a given language and a description of the way in which the procedures appearing in a given structure are executed.

A system that is able to perform the task of understanding will be called a language understanding system (for short L.U.system)

Definition 0.1. A *language understanding system S* is defined by a quadruple $S = \langle G, \Pi, P, \gamma \rangle$ where G is a grammar, Π is a parsing system accepting G, P is a set of procedures and γ is a function relating procedure names to procedures.

An L.U.system is such that Π is depending on the type of grammar being used while once the tipe is fixed G is variable to the system.

Definition 0.2. The *source language* for a given L.U.system is the language being accepted by the system.
This involves the fact that the parser is capable of analysing by means of the grammar all combinations of the source language and that P conteins procedural definitions for all procedure names in the lexicon of the language.

1. CLOSED COMPLETION GRAMMARS

1.1. Basic definitions

Definition 1.1. A *closed completion grammar* is a quadruple $G = \langle Voa, Vha, Vp, \delta \rangle$ where

Voa is a finite nonempty set of arguments called the set of occurred arguments,

Vha is a finite nonempty set of arguments called the set of hidden arguments,

$Voa \cup Vha = Va$ and Va is called the set of arguments,

Vp is a finite nonempty set of procedure names and $Vp \cap Va = \emptyset$,

δ is a finite set each element of which is a finite ternary relation included in $Va^{\star} \times Vp \times Vha$, relating arguments to procedures.

If $\langle \gamma, A, a \rangle \in \delta$ where $\gamma \in Va^{\star}$, $A \in Vp$ and $a \in Vha$, then we write $\gamma \rightarrow A \rightarrow a$. $\gamma \rightarrow A \rightarrow a$ is called a rule. The arguments appearing on the left of a rule are called the input arguments and the arguments appearing on the right of the rule the output arguments.

Example 1.1. Let $G = \langle Voa, Vha, Vp, \delta \rangle$ be a closed completion grammar where $Voa = \{a,b,c,d\}$, $Vha = \{e,f,g\}$ and $Vp = \{A, B, C\}$ and δ :

1. a b $\rightarrow$ A $\rightarrow$ e
2. e f c $\rightarrow$ B $\rightarrow$ g
3. d $\rightarrow$ C $\rightarrow$ f

A closed completion grammar G describes a language (called L(G)) in the following way. Starting with an arbitrary hidden argument, replace it by a procedure name of which this argument is output and add all the input arguments to the combination. If there is a hidden argument among these arguments, again replace it by a procedure of which these argument is output and add all the input arguments to the combination. If after a finite number of steps all elements are either procedure names or occurred arguments, the combination is complete.

More formal:

Definition 1.2.

(i) If there is a combination x u y (x,y possibly empty) where $x,y \in (Va \cup Vp)^{\star}$ and $u \in Vha$ and if there is a rule in the grammar $a_1 \ldots a_n \rightarrow A \rightarrow u$ $(n \geq 1)$ where $a_1, \ldots, a_n \in Va$ and $A \in Vp$, then we say that x u y *Preferentially directly derives* $x\ A\ a_1 \ldots a_n\ y$, denoted as $x\ u\ y \Longrightarrow x\ A\ a_1 \ldots a_n\ y$.

(ii) Let $\overset{\star}{\Longrightarrow}$ be the transitive reflexive closure of the relation $\Longrightarrow$. If $x \overset{\star}{\Longrightarrow} y$ then we say that x *preferentially derives* y.

(iii) The language of G, denoted as L(G) is defined by $L(G) = \{ y \mid y \in (Voa \cup Vp)^{\star}$ and $x \overset{\star}{\Longrightarrow} y$ where $x \in Vha \}$.

Example 1.2. Let G be the closed completion grammar of example 1.1. then the following derivations are possible. (The index on $\Rightarrow$ is the applied rule of the grammar)

(i) $f \overset{3}{\Rightarrow} C\ d$

(ii) $e \overset{1}{\Rightarrow} A\ a\ b$

(iii) $g \overset{2}{\Rightarrow} B\ e\ f\ c \overset{1}{\Rightarrow} B\ A\ a\ b\ f\ c \overset{3}{\Rightarrow} B\ A\ a\ b\ C\ d\ c$

Example 1.3. Let $G = \langle Voa, Vha, Vp, \delta \rangle$ be a closed completion grammar and $Voa = \{a,b,c,d\}$, $Vha = \{e,f\}$ $Vp = \{A, B, C, D\}$ and δ :

1. a b e → A → e
2. c e f → B → f
3. d → C → e
4. d → D → f

Some derivations:

(i) $f \overset{2}{\Rightarrow} B\ c\ e\ f \overset{1}{\Rightarrow} B\ c\ A\ a\ b\ e\ f \overset{3}{\Rightarrow} B\ c\ A\ a\ b\ C\ d\ f \overset{2}{\Rightarrow} B\ c\ A\ a\ b\ C\ d\ B\ c\ e\ f$
$\overset{1}{\Rightarrow} B\ c\ A\ a\ b\ C\ d\ B\ c\ A\ a\ b\ e\ f \overset{3}{\Rightarrow} B\ c\ A\ a\ b\ C\ d\ B\ c\ A\ a\ b\ C\ d\ f$

(ii) $f \overset{4}{\Rightarrow} D\ d \overset{4}{\Rightarrow} B\ c\ A\ a\ b\ C\ d\ B\ c\ A\ a\ b\ C\ d\ D\ d$

(iii) $f \overset{2}{\Rightarrow} B\ c\ e\ f \overset{1}{\Rightarrow} B\ c\ A\ a\ b\ e\ f \overset{3}{\Rightarrow} B\ c\ A\ a\ b\ C\ d\ f \overset{4}{\Rightarrow} B\ c\ A\ a\ b\ C\ d\ D\ d$

(note that L(G) is infinite)

In our definition of a direct derivation there is something that needs a bit more explanation, namely the word preferential,

It is well known that the formal theory of languages, dealing in particular with the so called Chomsky or phrase structure grammars, their related automata and their possible augmentations, has been exclusively based on strings formed by the operation of concatenation. Indeed the essence of these systems is that they define a strict linear order on the elements of a language and 'grammatical' means that a particular order is present.
What we propose here is to consider language utterances not as strictly ordered as it is usually done, rather we will introduce the concept of preferential order, being an order which is most likely to occur. In this respect the occurrence itself of an element is more important than the moment when it occurs.

We hope to gain by this approach not only a greater flexibility, a possible cure for the britleness of current natural language processing systems, but also a means of dealing with other levels of language than syntax and morphology, notably those where order is not as relevant as occurrence.

In linguistic theory strings are defined as objects consisting of an ordered set of occurrences of the elements of an alphabet. Now we introduce a 'weaker' object, called a combination, where the order is not so relevant anymore.

Definition 1.3. Let Σ be a finite alphabet then a *combination* over Σ is a set of occurrences or tokens of the elements of Σ .

Notation: As the distinction between combinations and strings is relative to the point of view, combinations will be written as strings.
Example 1.4. Let $\Sigma = \{a,b,c\}$ then examples of combinations are a b c , a b , a a b c , etc... .

From the definitions it follows that if a combination is considered to have a particular ordering (e.g. the precedence order) then the combination will be called a string. E.g. if σ = a b c is considered to be a combination then σ = a b c = b a c = c a b , etc..., whereas if σ is considered as a string a b c $\neq$ b a c $\neq$ c a b ... .

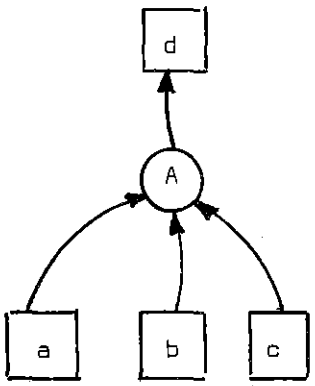
Let us now study the implication for our definition of the language . Recall that we defined the language of a completion grammar G as $L(G) = \{x \mid x \in (Voa\ Vp)^*$ and $y \overset{*}{\Rightarrow} x$, where $y \in Vha\}$. This language we will call the preferential language of G. To have a mathematical way of talking about nonpreferentially obtained strings, we introduce the concept of the associated language of G called $\widetilde{L(G)}$ and $\widetilde{L(G)} = \{y \mid x \in L(G)$ and y is a permutation of x $\}$.

So what we mean by 'is preferentially derived from' is that the precedence order imposed by this relation was preferential and in producing or generating this order is the goal. However if this order is not present because of a failure in the production or by influence of higher language levels (e.g. pragmatics), the analysis system does not block, as would be the case for phrase structure grammars. Also in cases of ambiguity, the order most approaching the preferential order will be the one chosen as the right analysis.
The problem in making these decisions is one of parsing and we will with this in 1.2..

Let us now discuss the format of the structures assigned by the grammar. These will not be labelled plane rooted trees or constituent structure trees, but a formally distinct structure to be introduced in this section. The significance of taking another format for the structures assigned by the grammar should not be underestimated. The validity will follow from its usage.

We reprent procedures as circles, called procedure circles, with the name of the procedure in it, and arguments as squares called argument squares. The input and output relations will be represented by directed lines from the arguments to the procedure circles. These lines can be labelled if there is any need to do so.

Example 1.5. Let a,b,c be input arguments and d output argument for a given procedure A, then this information is represented as follows:



The whole graph is called a relation structure, because it represents the functional relations among the elements.

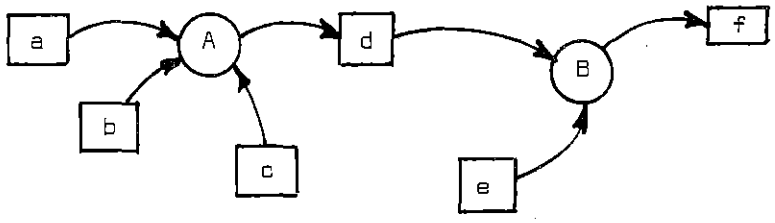
Definition 1.4. A *relation structure* is a construct ⟨Vp,Va, R⟩ where Vp is a set of procedures, Va is a set of arguments, R $\subseteq$ (Va x Vp) $\cup$ (Vp x Va) is a set of ordered pairs describing input relations (Va x Vp) and output relations (Vp x Va).

In this paper we will not investigate formally relation structures and for ease of discussion alway use the graphic representation.

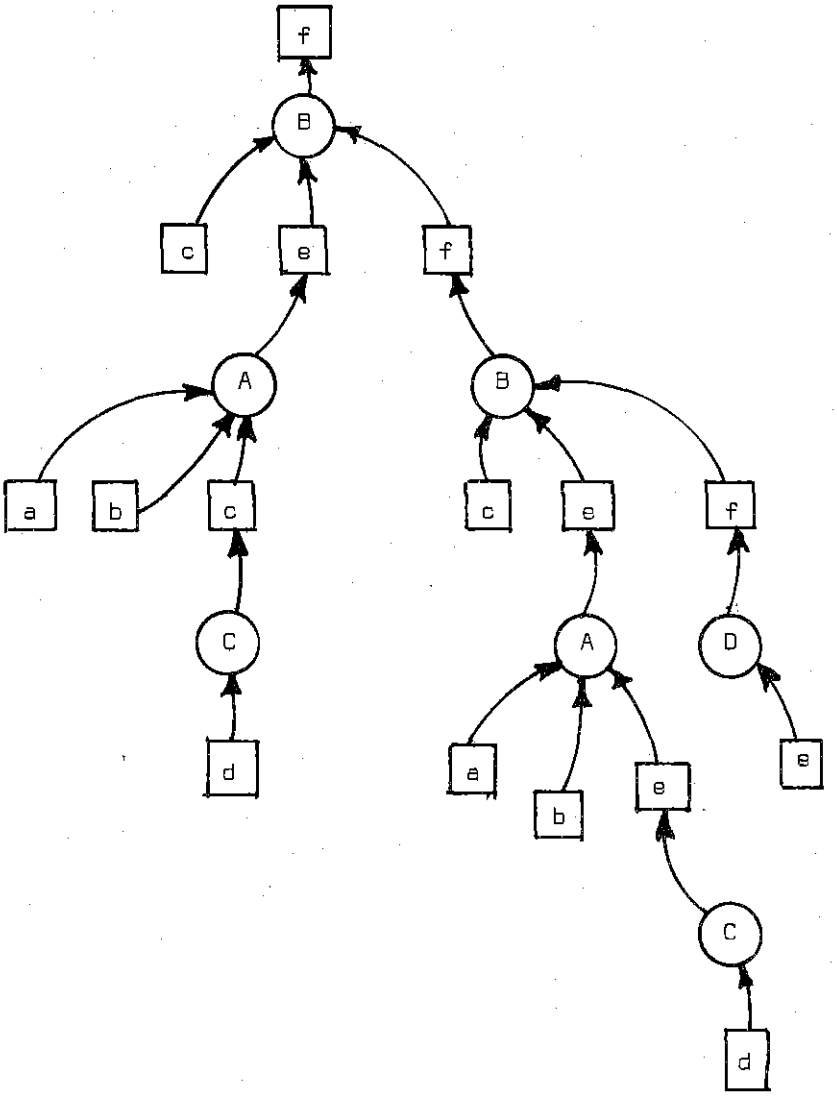
Let us now define relation structures in relation to combinations.

Definition 1..5 The *relation structure for a given combination* is a set of procedure circles representing the procedures in the combination ,a set of argument squares representing the arguments occurring in the combination itself or as an output argument of a procedure and a set of directed lines between the squares representing input and output relations.
Clearly a line leaving a circle is denoting an output relation whereas a line leaving a square is denoting an input relations.

Example 1.6. Let a, b, c be input arguments and d output argument for a given procedure A, and let d and e be input arguments for a given procedure B where f is the output argument then the relation structure for the combination A a b c B e is:



It is easy to see how to obtain relation structures (as defined in definition 1.5.) during the derivation process. Given an hidden argument as output, draw a square for it, connect it with an arrow to the procedure circle and for all input arguments draw squares and make connections to the procedure circle. For derivation (i) in example 1.3.) this would result in the following structure:



Definition 1.6. A completion grammar is *deterministic* if for each procedure in Vp there is one and only one rule in the grammar.
A completion grammar is *nondeterministic* if there is more than one rule for the same procedure.

All examples up to now were examples of deterministic closed completion grammars.

Example 1.7. Let G = ⟨ Voa, Vha, Vp, δ ⟩ be a closed completion grammar and Vha = {a,b,c}, Voa = {e,d,f}, Vp = {A} and δ:

1. a d → A → c
2. e f → A → b
3. b → A → a

Clearly G is nondeterministic.

PROBLEMS 1.

(i) Are the combinations d D, A a b e , B c C d D d in the language generated by the grammar of example 1.3. ?
(ii) Construct other closed completion grammars and generate some combinations.

1.2. *The parsing problem for closed completion grammars*

In linguistic science, a recognizer is a system that takes a grammar and an input string and decides whether or not the string is in the language (supposed to be) described by the grammar or not.

A parser on the other hand is a system that takes a grammar and an input string and produces the structural description assigned to this input string by the grammar. Of course if the input is ungrammatical there can be no structural description, so a parser implies a recognizer (but not vice-versa).

Let us now deal with the parsing problem for closed completion grammars by giving an algorithm that solves the problem. Due to spacelimitations, we will only deal with deterministic closed completion grammars here.

Algorithm 1.1. Let there be a pushdown stack (for short pds.) T1 where all procedures are stored and a pds. T2 for all arguments. Although in a concrete implementation the (partial) relation structure is stored in a list structure or a table representation of a list structure, for the sake of clarity in the exposition we will here use a graphic representation. Let σ be a given input combination and σ_i the i-th element of the combination.

Scan the input from left to right:

A. if σ_i is a procedure:
   1. create a procedure circle in the structure and put the procedure on T1.
   2. (a) check whether there are any arguments on T2 which can be input to the procedure according to the grammar, if so connect and take that particular argument from the pds. T2.
      (b) if all input arguments are found (we say that the procedure is complete) remove the procedures from T1, put the output element as argument square in the structure and connect it with an output relation to the procedure circle; then execute the B.2. part of this algorithm.

B. if σ_i is an argument:
   1. Create an argument square in the structure
   2. Check for all procedures on T1 whether this argument can be input to it. If so connect, else put it on T2. If the procedure is complete, do the same as was specified under A.2.(b) of this algorithm.

To be grammatical there should be one and only one element on T2 and none on T1 after scanning the whole input. The final element on T2 is the initial point in the derivation.

Example 1.8.

Let the grammar be G =(Voa, Vha, Vp, δ) and Voa = {a,b,c,d},
Vha = {e,f} Vp = {A, B, C, D} and δ :

1. a b e → A → e
2. c e f → B → f
3. d → C → e
4. d → D → f

Derivation 1:

$e \overset{1}{\Rightarrow} A\ a\ b\ e \overset{3}{\Rightarrow} A\ a\ b\ C\ d$

σ = A a b C d

(i) σ_1 = A

1. 

(create a procedure circle in the structure and put the procedure on T1)

T1: A

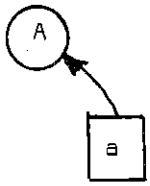
2. T2 is empty, no checking.

(ii) σ_2 = a

1. 

(create a square in the structure)

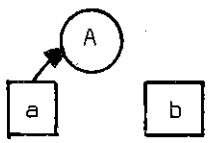
T1: A
T2: -



(according to the grammar a is input to A, so we connect a to A)

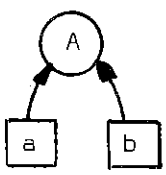
T1: A
T2: -

(iii) σ_3 = b

1. 

(create a square in the structure)

T1: A
T2: -

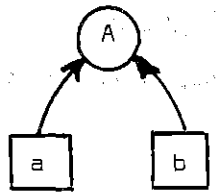
2. 

T1: A
T2: -

(according to the grammar b is input to A, so we connect b to A)

(iv) σ_4 = C

1.



(create a procedure circle for C and put C on T1)

T1 : C A

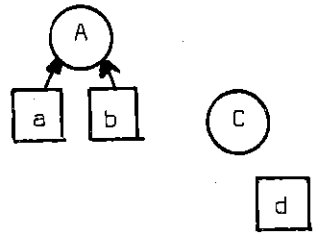
T2: -

2. T2 is empty: no checking

(v) σ_5 = d

(create an argumentsquare in the structure)

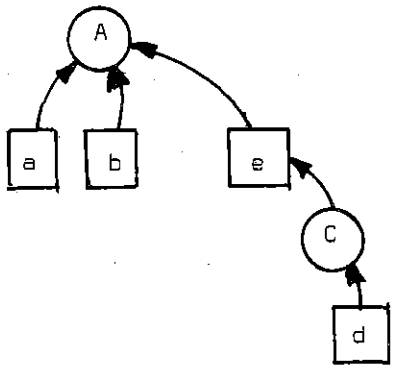
1.



T1: C A

T2: -

2.

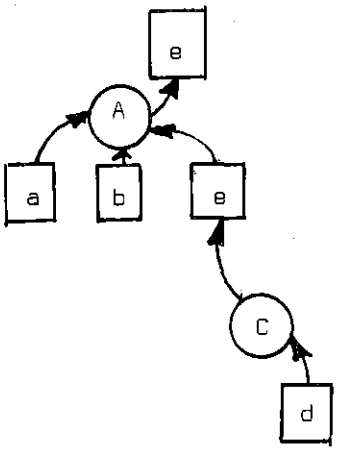


(As d is input for C according to the grammar, we connect d to C. By this C is complete and we add the output of C to the structure. This output is according to the grammar input to A, hence we make a connection to A)

T1: A

T2: -

3.



(By adding the output of C , i.e. e, to the structure also A is complete)

T1: -

T2: e

Note that the final element on T2 is the initial point of the derivation.

Derivation 2.

f $\overset{2}{\Rightarrow}$ B c e f $\overset{1}{\Rightarrow}$ B c A a b e f $\overset{3}{\Rightarrow}$ B c A a b C d f $\overset{4}{\Rightarrow}$ B c A a b C d D d

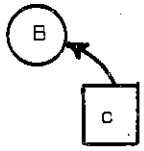
At each step we now give only the partial structure and the contents of T1 and T2.

σ = B c A a b C d D d

(i) σ_1 = B

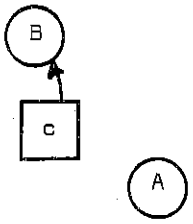
T1: B
T2: -

(ii) σ_2 = c



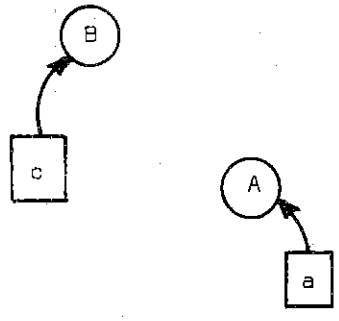
T1: B
T2: -

(iii) σ_3 = A



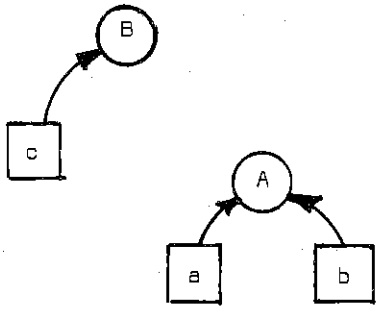
T1: A B
T2: -

(iv) σ_4 = a



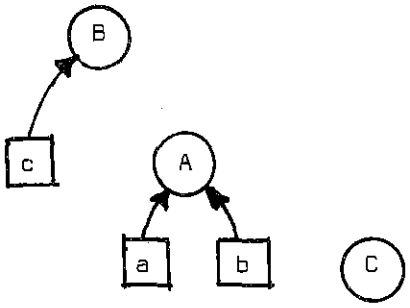
T1: A B
T2: -

(v) σ_5 = b



T1: A B
T2: -

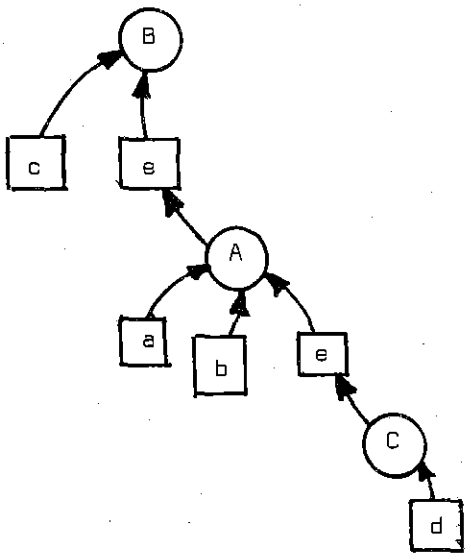
(vi) σ_6 = C



T1: C A B

T2: -

(vii) σ_7 = d



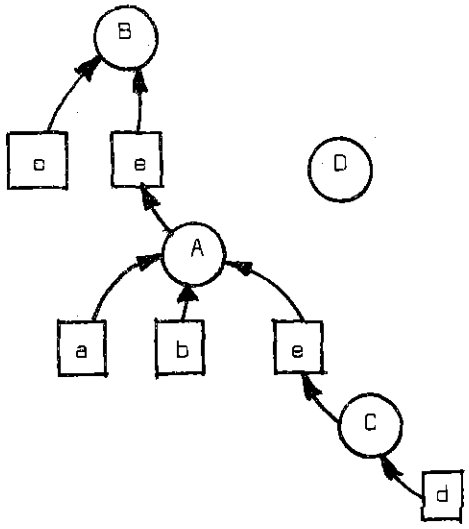
T1: B

T2: -

(C is complete by d and A is complete by adding the output of C)

(viii)

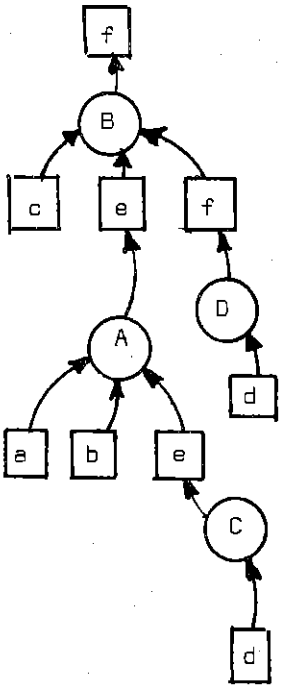
σ_8 = D



T1: D B

T2: -

(ix) σ_9 = d



T1: -
T2 : f

(D is complete and B is complete by output of D)

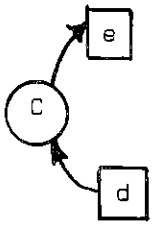
Derivation 3.

In the next example we show what happens with an input combination which is not in a preferential order. Let us take the reverse of the combination obtained by derivation 1, namely σ = d C b a A

(i) σ_1 = d

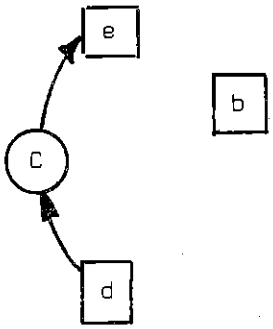
T1: -
T2: d

(ii) σ_2 = C



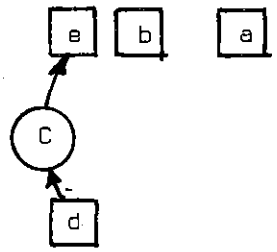
T1: -
T2: e

(iii) σ_3 = b



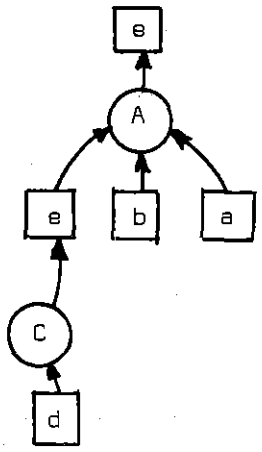
T1: -
T2: b e

(iv) σ_4 = a



T1: -
T2: a b e

(v) σ_5 = A



T1: -
T2: e

The reader is advised to compare this parsing process with the one used for a preferential order on this combination, and to perse other orderings over this combination. He will see that the same result is obtained. Some combinations will lead to a very clumsy parsing process. The easiest parsing will be the one with a preferential precedence order on the input combinations.

PROBLEMS 2.

(i) Construct a program for algorithm 1.1. in an available programming language and test the examples given.

(ii) Let G = ⟨ Voa, Vha, Vp, δ ⟩ be a closed completion grammar where Voa = {a,b,c} Vha = {d,e} , Vp = {A, B, C} and δ :

d → A → d

a b c e → B → e

b e → C → e

Describe the language generated by this grammar.

(iii) Let G =⟨ Voa, Vha, Vp, δ ⟩ be a closed completion grammar where Voa = {a,b,c} Vha = {d,e}, Vp = {A, B, C} and δ :

1. a b e → A → d
2. d b c → B → d
3. b → C → e

Parse the following examples with algorithm 1.1.: (i) A a b C b (ii) b C b a A (iii) B A a b C b b c

Before we deal with the application of completion grammars two important remarks should be made on the nature of the arguments:

1. Contrary to intuition, we think that there are no real or 'occurred' arguments appearing in any language input itself. Some examples will make this clear.

Take the sign '12', it may be thought that '12' is a simple argument for an arithmetic procedure or so, however understanding '12' involves a computation based on the decimal number system: $1 \times 10^1 + 2 \times 10^0$. So, although '12' does not take any input arguments, it implies a procedure to which it is input.

As another example take the sign 'p' as it is used in the propositional calculus, i.e. a propositional variable. Now again understanding 'p' involves a procedure: checking whether 'p' has already a value and if not store a new variable name of yet unknown value.

Similarly a pronoun involves a procedure computing the reference of the pronoun a proper name involves checking where the name appears in the memory (data base), etc...

So, what one normally thinks to be simple arguments are arguments for a procedure that is supposed to be known by the understander. For ease of discussion we will from now on treat these objects as arguments appearing in the combination itself, and call them occurred arguments as opposed to hidden arguments.

2. Although arguments were represented (in a formal treatment) by single letters, they have in fact an internal structure, in particular an argument has an argument value , an argument type and an argument name , that is a sign by which a particular argument is denoted.
When an argument has not yet a value it is called a dummy argument.
When an argument has not yet a name, it is called an anonymous argument.

E.g. when talking about the variable I2, we could say that it is:

(i) an integer (argument type)
(ii) called I2 (argument name)
(iii) having e.g. the value 20 (argument value)

We have now another way of making the distinction between hidden and occurred arguments: all hidden arguments are anonymous and all occurred arguments are not anonymous.

1.3. *Application to the P.C.language*

We have now reached the point where we can put this formal framework to use. We will do this by discussing a language which is certainly known to anyone and which has such properties that one can deal with it by means of closed completion grammars. The language we have in mind is the simple propositional calculus (for short PC-language) in Polish notation.
We hope that by giving a fully worked out example the reader will see the relevance of our approach and is encouraged to read on.

(a) Current descriptions of the language.

A *logician* would define the SYNTAX of the PC-language as follows.
Let there be a set of propositional operators: { NOT, AND, OR, IMPLIES, EQUIVAL } and a set of propositional variables or elementary propositions (i.e. propositional variables having a truth value), then
1. Every elementary proposition or propositional variable is a propositional expression.
2. Every propositional expression preceeded by NOT is a propositional expression.
3. Every combination of two propositional expressions by means of one of the other propositional operators is a propositional expression.
4. The PC-language consists solely of propositional expressions.

Examples: NOT p, AND p q, IMPLIES p q , etc...

A *linguist* would define the syntax of the PC-language as follows. Let
G = (Vn, Vt, P, EXPR) be a context-free grammar where Vn = { EXPR, OPER },
Vt = { AND, OR, IMPLIES, NOT, EQUIVAL, p,q,r,... } and P:

1. EXPR → OPER EXPR EXPR
2. EXPR → NOT EXPR
3. EXPR → p , q , r , ..
4. OPER → AND, OR, IMPLIES, EQUIVAL

This way of defining the language has the advantage that a structure can be recognized in a propositional expression and this helps when calculating truth values.

(b) Closed completion grammars for the PC-language

How should we deal now with the PC-language from a completion grammars point of view. First of all, we make a distinction between procedures and arguments. Procedures are clearly NOT, AND, OR, IMPLIES, EQUIVAL. We add also the function SET by which one can assign a truth value to a propositional variable (e.g.'SET P TRUE')and the function '?' by which one can ask the truth value of a propositional variable or expression (e.g. '? P', '? AND P Q') .

The grammar is the following one. Let G = ⟨ Voa, Vha, Vp, δ ⟩ be a closed completion grammar where Voa = {LOG}, Vha = {LOG} Vp = {AND, OR, NOT, IMPLIES, EQUIVAL, SET ,? }

and δ

1. LOG LOG → AND → LOG
2. LOG LOG → OR → LOG
3. LOG → NOT → LOG
4. LOG LOG → IMPLIES → LOG
5. LOG LOG → EQUIVAL → LOG
6. LOG LOG → SET → LOG
7. LOG → ? → LOG

It is easy to see that one can abstract patterns and make the grammer simpler. This can be done by using more than one possible instantiation of the procedure name in a rule. In this way classes of procedure names can be definied.

X_1 = {AND, OR, IMPLIES, EQUIVAL}

X_2 = {NOT, ? , SET}

The patterns:

1. LOG LOG → X_1 → LOG
2. LOG → X_2 → LOG

Warning: The symbols X_1, X_2 should not be considered as a sort of nonterminals as one, used to the p.s.grammars framework,might be tempted to do.

Note also that LOG is the argument type . The argument name can be anything, e.g. TRUE, FALSE, P, Q, R, ... or even no name (for anonymous arguments), and the argument value is either assigned by means of the set-function or fixed as for TRUE which is always true. The value is of course either true, false or unknown.

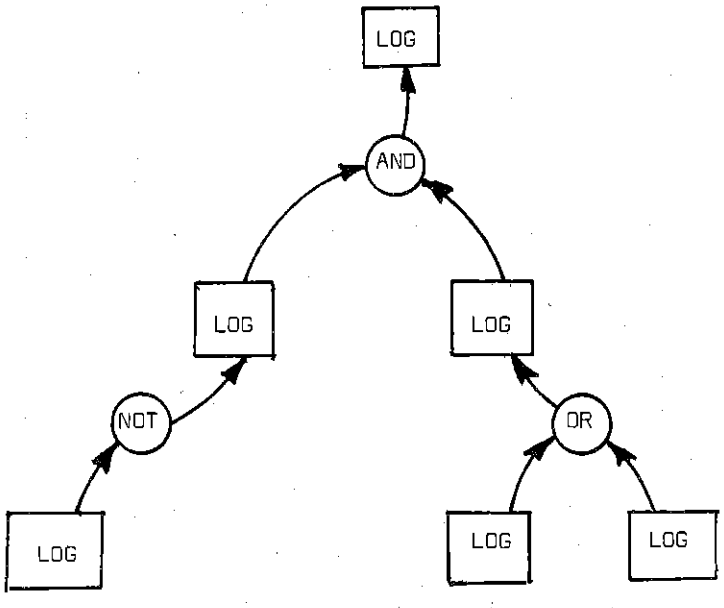
Some derivations:

(i) LOG ⇒ AND LOG LOG ⇒ AND NOT LOG LOG ⇒ AND NOT LOG OR LOG LOG

(ii) LOG ⇒ IMPLIES LOG LOG

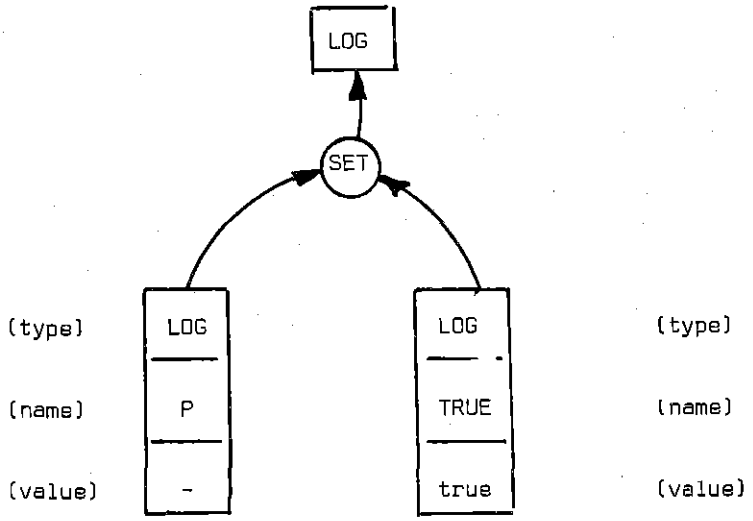
(iii) LOG ⇒ SET LOG LOG ⇒ SET LOG NOT LOG

The relation structure for derivation (i) is:

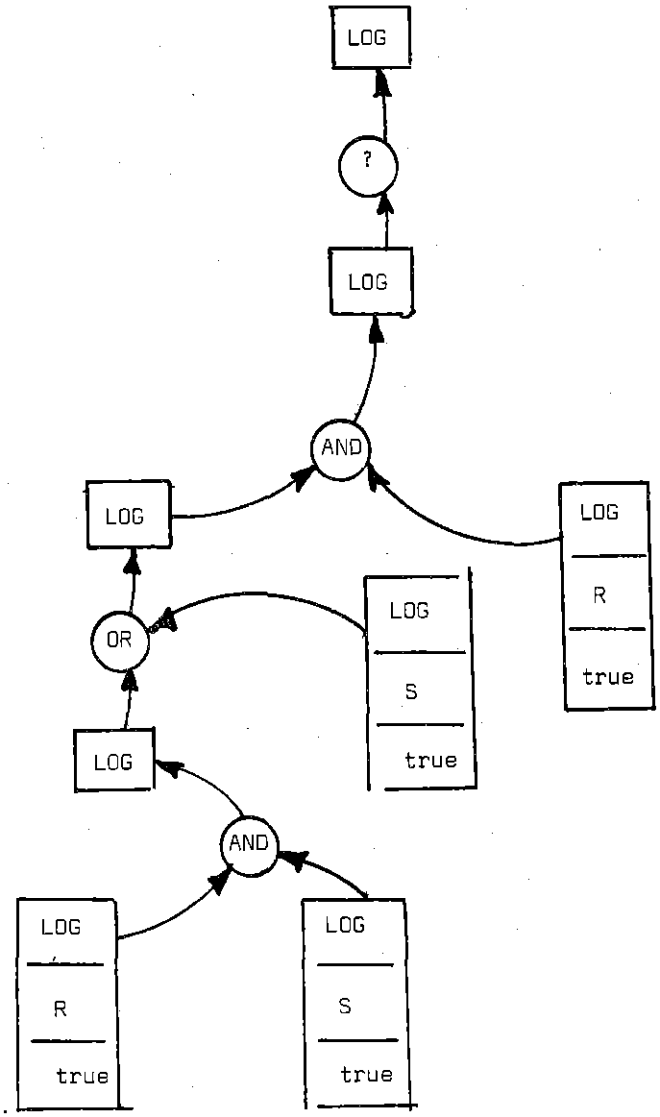


Example 1.9.

(i) SET P TRUE



(ii) ? AND OR AND R S S R (Where R and S are true)



1.4. The interpretation problem.

Recall definition 0.1. where an L.U.system was defined as a quadruple $S = (G, \Pi, P, \gamma)$. In previous sections we specified G, in particular a closed completion grammar, and Π the parser, in particular algorithm 1.1.. In this section we briefly discuss P and γ. Briefly, because this paper concentrates on analysis rather than interpretation. Indeed, interpretation should be more sophisticated then we will present it here.

(a) The set of procedure P.

A *logician* would deal with the SEMANTICS in the following way:
If p and q are propositional expressions, AND p q is true just in case both p and q are true, otherwise false.
If p and q are propositional expressions OR p q is true just in case p is true or q is true of both; otherwise it is false.
If p is a propositional expression, then NOT p is true just in case p is false, otherwise it is true.
If p and q are propositional expressions, then IMPLIES p q is false when p is true and q is false, otherwise it is true.
If p and q are propositional expressions then EQUIVAL p q istrue when p is true and q is true, or p is false and q is false, otherwise it is false.

Linguistics semantics is currently still a matter of debate and the procedural view which is basic to the approach presented in the following paragraph is not yet accepted by the whole linguistic community.

Let us give some procedures for the predicates of the propositional calculus. Let true be denoted by 0 and false by 1.
(There are other solutions possible)
(i) AND: if the sum of the values of the input arguments is 0 the value of the output-argument is 0, else it is 1.
(ii) OR: if the sum of the input values is smaller or equal to 1, the output value is 0, else 1.
(iii) IMPLIES: if the value of the second input argument minus the value of the first one is equal to 1, the output is 1, else 0.
(iv) EQUIVAL: if the input values are equal, the output is 0, else it is 1.
(v) NOT: the output value is 1 minus the input value.
(vi) SET: store the value of the first input argument in the value place of the second argument and set the output equal to this value.
(vii) ?: print the value of the input argument.

(b) The interpretation mechanism.

There are in general two ways of doing semantic interpretation or in other words organizing the subroutine calling the procedural definitions of the predicates and connecting them to the input arguments according to the relation structure.

Definition 1.7. An interpretation process is said to be *instant* if procedures are executed as soon as this is possible during the parsing process.
An interpretation process is said to be *delayed* if procedures are executed when the complete structure is available, or in other words after the parsing process.

The distinction between instant and delayed mode is very important. Not so much for closed completion grammars, but we will see that with open completion grammars, to be introduced in next sections, different structures (and thus interpretations) are obtained depending on whether the mode is instant or delayed. Clearly the difference between instant and delayed interpretation is related to the compiler/interpreter distinction known from translators of programming languages.

Definition 1.8. A procedure is said to be *instant* if it must be executed as soon as all its arguments are found in the input.
A procedure is said to be *delayed* if it is executed after the parsing process is completed.

When interpreting in instant resp. delayed mode, all procedures must be instant resp. delayed. Also it is possible to organize a mixed interpretation process, where instant and delayed procedures occur .

Let us now give algorithms for interpretations. As the instant mode is the easiest one, we deal with it first.

Algorithm 1.2. γ in instant mode.

As soon as a procedure is complete, i.e. if the perser has discovered all the input arguments, execute it.

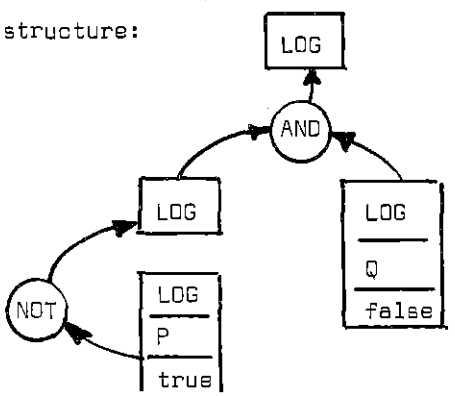
Algorithm 1.3. γ in delayed mode.

We start by the argument left on T2 ,i.e. the initial argument in the derivation, then we go to the procedure circle for which this argument was the output. For all input arguments of this procedure, check whether they are hidden or occurred. If hidden apply recursively γ , else goto the next argument. If all arguments are worked out in this way, execute the procedure.

Example 1.10.

' AND NOT P Q ' (where P is true and Q is false)

the structure:



We start with the uppermost LOG-square, go to the AND-procedure circle and check for every argument whether it is hidden or occurred. The first argument is hidden,so we start again with this one, go to the NOT-procedure circle and check its input arguments. This time the only input argument is an occurred argument; hence we execute the function NOT; Now we know the value of the first input argument of the AND-procedure and because the second input argument is an occurred argument we execute the AND-function.
For example 1.10. the final result would be false.
There is a lot more to say about techniques for carrying out semantic interpretation but this will do for the moment.
It is a good custom of scientists to do experiments. In this spirit we programmed algorithm 1.1. and 1.2. in FORTRAN IV and realised an implementation on the PDP 11/45
The performance of the system is illustrated by the following output.

In general for all experiments we use the following conventions for communicating with the system.

When the sign '?' appears, an input expression is being typed. The system will first return this input, preceeded by 'input ' and then start processing the expression. For ease of reference, the system automatically numbers the input according to their occurrence. Systems output is preceeded by 'out:'.

Example:

```
?                                   (request for input)
SET P TRUE                          (input expression of user)
INPUT  1: SET P TRUE                (system returns the input)
                                    (no output produced)

?                                   (request for input)
? P                                 (input expression of user)
INPUT  2: ? P                       (system returns the input)
OUT: TRUE                           (result of processing)
```

The following additional commands are accepted: +GRAMMAR (returns the grammar), +LEXICON (returns the lexicon) +END INPUT (means end of input stream).
There is a switch to ask for additional parsing information +STRUCTURE is for on and +NO STRUCTURE is to put it off.
Also all conventions for editing via teletype (e.g. RUBOUT) can be used.

```
?
SET P TRUE
INPUT  1: SET P TRUE
?
+GRAMMAR
 1.    3 10 10 10
 2.    3 10 10 10
 3.    3 10 10 10
 4.    2 10 10
 5.    3 10 10 10
 6.    2 10 10
 7.    2 10 10
 8.    3 10 10 10
 9.    3 10 10 11
?
+LEXICON
 1. AND        1  1  0  0
 2. OR         2  2  0  0
 3. IMPLIES    3  3  0  0
 4. NOT        4  4  0  0
 5. EQUIVAL    5  5  0  0
 6. ?          6  6  0  0
 7. PROOF      7  7  0  0
 8. SET        8  8  0  0
 9. (          9  9  0  0
10. LOG       10  0 10  2
11. TRUE      10  0 11  0
12. FALSE     10  0 12  1
13. )         11  0 13  0
14. P  10  0 14  0
?
SET Q FALSE
INPUT  2: SET Q FALSE
?
SET R NOT TRUE
INPUT  3: SET R NOT TRUE
?
+LEXICON
 1. AND        1  1  0  0
 2. OR         2  2  0  0
 3. IMPLIES    3  3  0  0
 4. NOT        4  4  0  0
 5. EQUIVAL    5  5  0  0
 6. ?          6  6  0  0
 7. PROOF      7  7  0  0
 8. SET        8  8  0  0
 9. (          9  9  0  0
10. LOG       10  0 10  2
11. TRUE      10  0 11  0
12. FALSE     10  0 12  1
13. )         11  0 13  0
14. P  10  0 14  0
15. Q  10  0 15  1
16. R  10  0 16  1
?
? P
INPUT  4: ? P
OUT: TRUE
?
? Q
INPUT  5: ? Q
OUT: FALSE
?
? AND P Q
INPUT  6: ? AND P Q
OUT: FALSE
?
? AND P AND Q R
INPUT  7: ? AND P AND Q R
OUT: FALSE
?
? AND P AND A AND ^U
? AND P AND Q AND Q R
INPUT  8: ? AND P AND Q AND Q R
OUT: FALSE
?
? OR Q R
INPUT  9: ? OR Q R
OUT: FALSE
```

Here we start our conversation with the system by the assignment of the value true to the propositional variable 'p'.

By giving in +GRAMMAR we can ask for the grammar (coded of course)

By giving in +LEXICON we can ask for the lexicon. Note that the variable 'p' which is initially unknown to the system has been added.

By INPUT 2 and INPUT 3 we introduce new variable names and values for them.

When asked for the lexicon again, one can see that 'q' and 'r' have been 'learned' so to say by the system.

Input 4 and 5 illustrate how the truth value can be asked for a simple variable

From input 6 onwards we give in some more complex expressions.

Here a typing mistake was made and corrected by means of the teletype conventions.

```
?
? IMPLIES Q R
INPUT 10: ? IMPLIES Q R
OUT: TRUE
?
? IMPLIES P Q
INPUT 11: ? IMPLIES P Q
OUT: FALSE
?
? IMPLIES Q P
INPUT 12: ? IMPLIES Q P
OUT: TRUE
?
? EQUIVAL IMPLIES P P IMPLIES Q Q
INPUT 13: ? EQUIVAL IMPLIES P P IMPLIES Q Q
OUT: TRUE
?
SET S NOT IMPLIES P Q
INPUT 14: SET S NOT IMPLIES P Q
?
? S
INPUT 15: ? S
OUT: TRUE
?
? NOT P
INPUT 16: ? NOT P
OUT: FALSE
?
? AND OR IMPLIES EQUIVAL P P Q R S
INPUT 17: ? AND OR IMPLIES EQUIVAL P P Q R S
OUT: FALSE
?
+STRUCTURES
?
? AND OR AND R S S R
INPUT 18: ? AND OR AND R S S R
OUT: FALSE
STRUCTURES :
-----------
NODES :
 1.    6  6  1  0  6
 2.    1  1  2  0  1
 3.    2  2  3  0  2
 4.    1  1  4  0  1
 5.   10  4  4  1 16
 6.   10  4  5  0 17
 7.   10  3  4  1 14
 8.   10  3  5  0 17
 9.   10  2  4  0 15
10.   10  2  5  1 16
11.   10  1  4  1 16
12.   10  0  0  1  0
RELATIONS :
 1.    2  0 12 11
 2.    3  0 11  9 10
 3.    3  0  9  7  8
 4.    3  0  7  5  6
T2 :  12
?
+NO STRUCTURES
?
P ?
INPUT 19: P ?
OUT: TRUE
?
P AND Q ?
INPUT 20: P AND Q ?
OUT: FALSE
?
P IMPLIES Q ?
INPUT 21: P IMPLIES Q ?
OUT: FALSE
?
P AND P IMPLIES P OR A ?
INPUT 22: P AND P IMPLIES P OR A ?
OUT: FALSE
```

Now we illustrate the structures switch, +STRUCTURES puts it on, and for all input-expressions from now on the relation structure (in a coded form) is produced.
For a graphic representation of this structure we refer to example 1.9.

By +NO STRUCTURES we put the switch off again.

From input 19 we start to experiment a little with other orderings over the input. Recall that the input expression is considered as a combination not a string. In this spirit also not preferentially ordered inputs must be processed. This is clearly the case, as one can see from the examples.

As there are no sophisticated error mechanisms, unknown variables will not necessarily block the interpretation process.

```
?
? A
INPUT 23: ? A
OUT: VALUE UNKNOWN
```

That the system knows very well that A is of unknown value is illustrated by input 23. Note that A has been introduced in the middle of an expression and not elsewhere.

```
?
P AND P IMPLIES P OR Q ?
INPUT 24: P AND P IMPLIES P OR Q ?
OUT: TRUE
```

From input 24 it is clearly to be seen that the mode of interpretation is instant, in fact the following expression is processed: (((p and p) implies p) or q)?

```
?
P AND Q AND P AND Q AND P AND R ?
INPUT 25: P AND Q AND P AND Q AND P AND R ?
OUT: FALSE
?
P AND Q OR P ?
INPUT 26: P AND Q OR P ?
OUT: TRUE
?
+STRUCTURES
```

Another illustration of the +STRUCTURES switch .

```
?
P AND Q OR P ?
INPUT 27: P AND Q OR P ?
OUT: TRUE
STRUCTURES :
------------
NODES :
 1.   10   2   4   0 14
 2.    1   1   1   0  1
 3.   10   2   5   1 15
 4.   10   5   4   1  1
 5.    2   2   2   0  2
 6.   10   5   5   0 14
 7.   10   8   4   0 14
 8.    6   6   3   0  6
 9.   10   0   0   1 15
RELATIONS :
 1.    3   0   4   1  3
 2.    3   0   7   4  6
 3.    2   0   7   7
T2 :   9
?
+NO STRUCTURES
?
P Q IMPLIES ?
INPUT 28: P Q IMPLIES ?
OUT: FALSE
?
Q P IMPLIES ?
INPUT 29: Q P IMPLIES ?
OUT: TRUE
?
P Q R AND OR ?
INPUT 30: P Q R AND OR ?
OUT: TRUE
?
P P EQUIVAL ?
INPUT 31: P P EQUIVAL ?
OUT: TRUE
?
P
INPUT 32: P
?
SET SET
INPUT 33: SET SET
UNGRAMMATICAL INPUT
?
AND P
INPUT 34: AND P
UNGRAMMATICAL INPUT
?
+END INPUT
```

Now we give in some expressions in postfix notation. They are all processed. Note that postfix is considered as the exact reverse of prefix as is illustrated by input 28 and 29

Input 32 is a combination, nothing is asked nothing is being returned, the system only computes the value of p.

The only point were ungrammaticality is noticed is with incomplete procedures.

1.5. Applications to natural language

Now we show that closed completion grammars can also be used as a model for (subsets) of natural languages. In particular we will investigate nominal phrases from this point of view. We do not present a fully worked out discussion here, only an indication of the direction in which more detailed research should proceed.

The examples will all be taken from Dutch, but an 'literal' English translation is provided. The universe of discourse for the experiments is the language of simple arithmetics. This is so because there are no complicated memory procedures (as storing or retrieving information) necessary. As the problem of memory organization is another (almost blank) page in the study of natural language behaviour, this universe of discourse is avoiding the problem, such that experimentation remains possible.

The basic hypothesis is of course that all elements in a noun phrase (nouns, determiners, adjectives, adverbs) are either procedures or arguments. Let us discuss very briefly how this would go.

(i) Nouns are either procedure names, either arguments.
(a) Arguments are such things as proper names, numbers, names for variables (e.g. the word 'number', 'person'), pronouns, etc...
(b) If a noun is a procedure then it takes other arguments as input. What arguments are input to the procedure denoted by a given noun depends on the argument type (as was the case for artificial languages) but also on additional information of a syntactic and morphological nature, i.c. prepositions or case endings. These will act upon the type of an argument.

Convention Whenever more than one specification concerning the type of the arguments that can be input appears in the grammar, we use square brackets and write all specifications in it separated by comma's.

Now we can given an example of a noun being a procedure name:

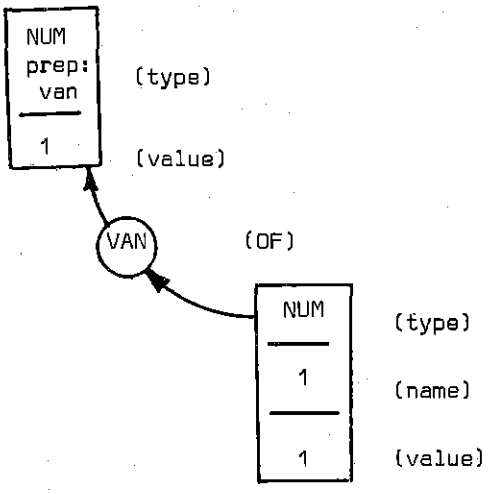
'De deling van 1 door 1'
('The division of 1 by 1')

The procedure is here 'deling' (division) , it takes two arguments both of a number type, however the first argument has the indication with preposition 'van' (of), and the second with preposition 'door'(by).
Rules in the grammar would look as follows:

(NUM, prep:VAN) (NUM,prep:DOOR) ⟶ DELING ⟶NUM
((NUM,prep: OF) (NUM, prep:by) ⟶ DIVISION ⟶NUM)

(ii) Prepositions seem to be procedures that add only a characteristic feature to the type of the output but do not change the value.

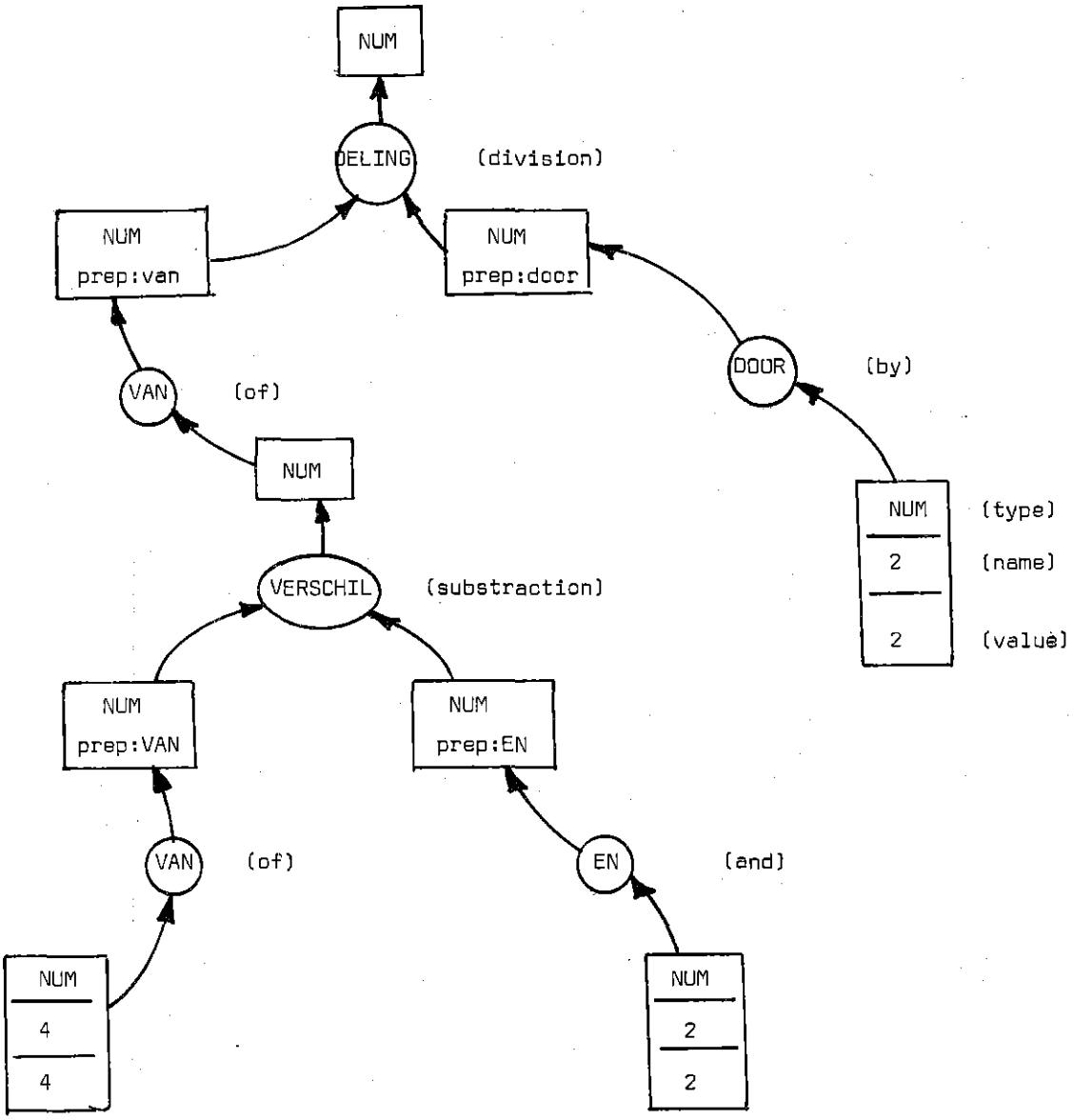
e.g.: 'VAN 1'
(of 1)



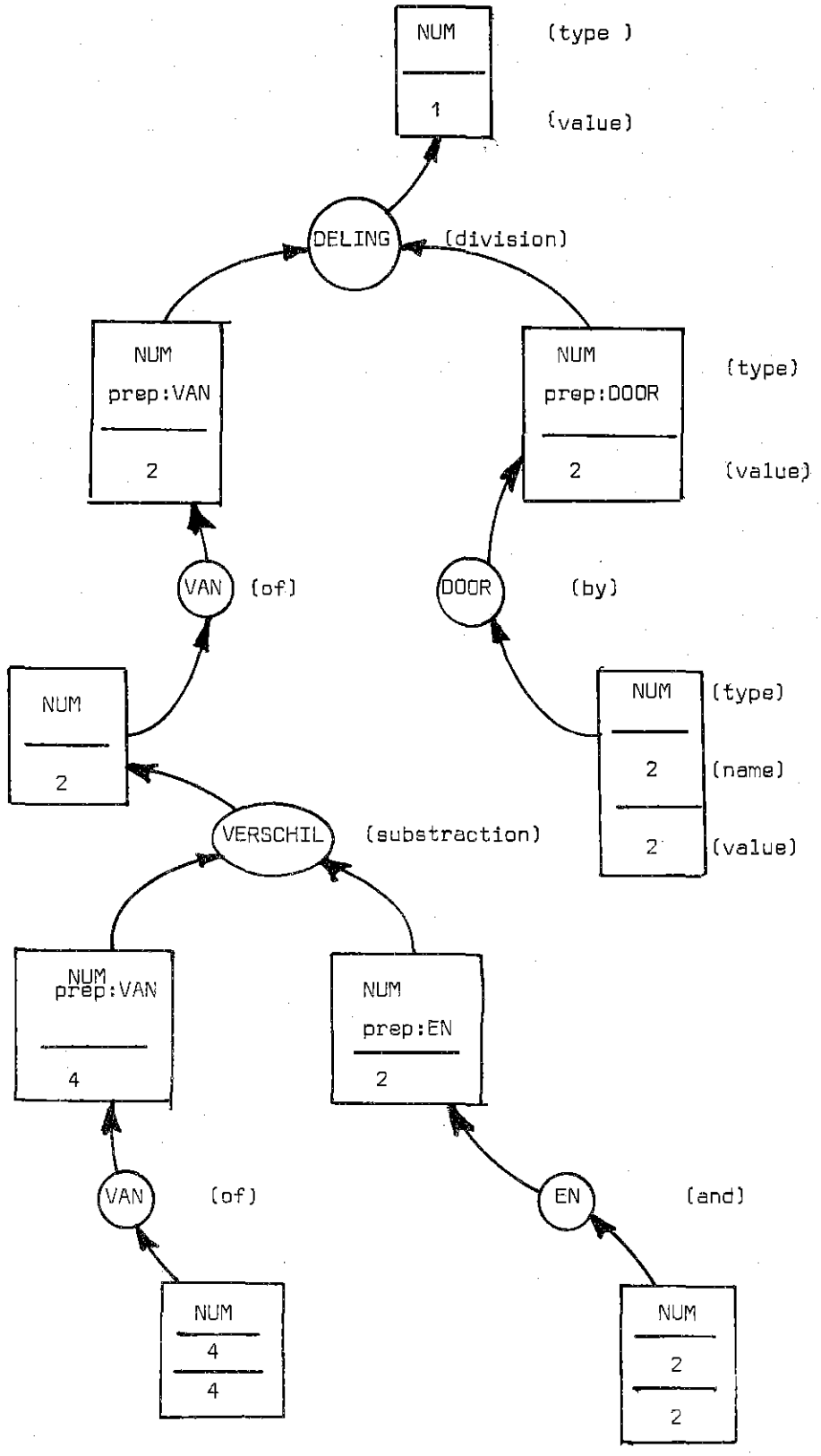
An important observation is that the proposition of a noun which is itself a procedure (rather than an argument as in the previous example) goes over to the output argument of that procedure.

E.g.: 'De deling van het verschil van 4 en 2 door 2'
(the division of the substraction of 4 and 2 by 2'

The relation structure:



After execution of all functions:



Clearly the final result is 1.

(iii) Plural endings of nouns seem to indicate the size of the output for a given procedure. Singular denotes one single element (as was the case in all examples up to now) or a set (seen as a whole) whereas plu 1 is an indication that more than one element is to be expected in the place of the output argument.

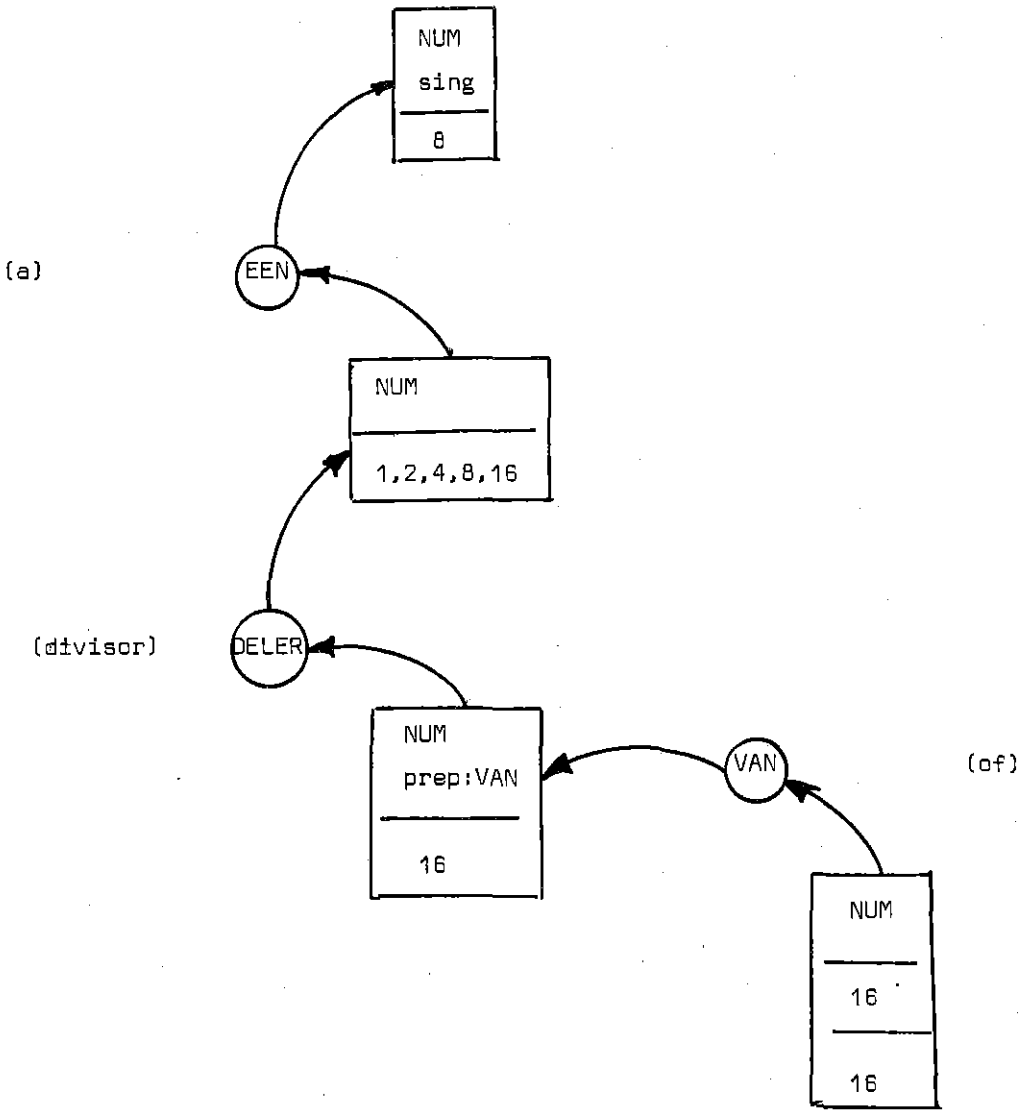
In this way singular/plural information acts as a sort of mechanism by which storage is provided for one or more elements (cf. dimension statement familiar from some programming languages).
We indicate this by adding plural or singular to the argument type of the output argument.

(iv) Determiners seem to organize 'loops' (in the programming sense) upon the execution of the noun phrase, or otherwise a final mechanism of selection acting upon the elements in the output argument.
E.g. 'an,a' : takes one arbitrary number of the set, if the set contains only one element than the choice is no more arbitrary.
'some': returns more than one arbitrary element of the set, etc..

E.g: 'Een deler van 16'
(A divisor of 16)

'Deler' (divisor) is a function computing all numbers by which another number can be divided. The divisors of 16 eg. are 1,2,8,4,16.

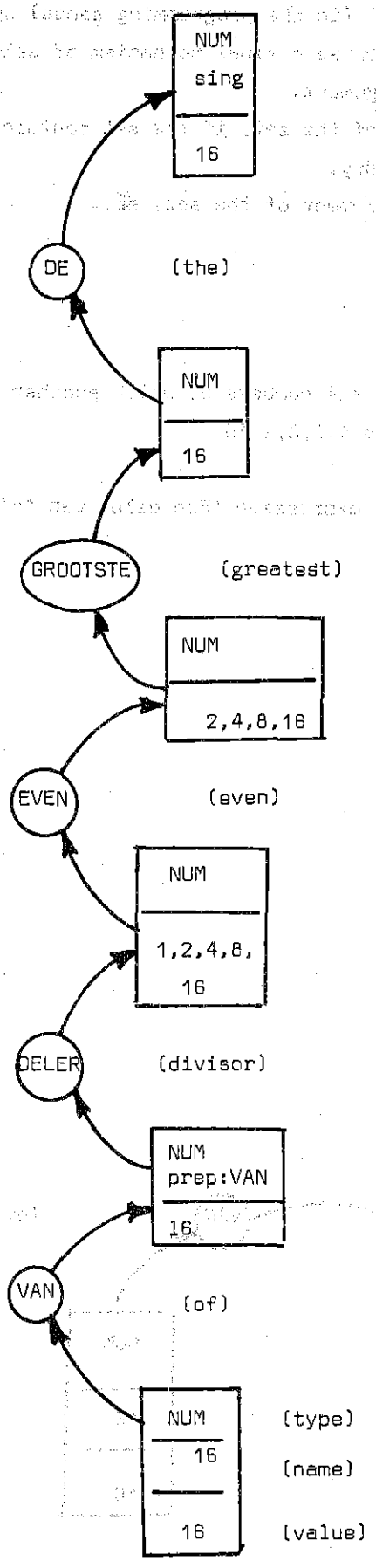
After execution of the procedures for the expression 'Een deler van 16' we get:



(iv) Finally adjectives and adverbs seem to be procedures that take the output of the noun as input and perform a further computation on this.

E.g. 'De grootste even deler van 16'
(the greatest even divisor of 16)

After execution of the functions the structure looks as follows:



We stress that all our remarks on the nature of the procedures are very tentative. It is not a subject of this paper and we only want to show how the underlying framework works. Let us now construct a full grammar for arithmetic expressions:

Let G =(Voa, Vha, Vp, δ) be a closed completion grammar where Voa = {NUM, Hoeveel, Wat}
Vha ={NUM, (NUM,prep:VAN),(NUM,prep:DOOR),(NUM,prep:EN)}
and Vp = {DE, HET, EEN, SOM, VERSCHIL, PRODUCT, DELING, DELER(S), GROOTSTE, KLEINSTE, EVEN , ONEVEN, ENKELE, VIERKANTSWORTEL, TWEEDEMACHTSWORTEL, ? }
and δ contains the following patterns:

1. NUM → X_1 → NUM
2. (NUM,prep:VAN) (NUM,prep: EN) → X_2 → NUM
3. (NUM,prep:VAN) (NUM,prep:DOOR) → X_3 → NUM
4. (NUM,prep:VAN) → X_4 → NUM
5. NUM → VAN → NUM,prep:VAN
6. NUM → EN → NUM,prep:EN
7. NUM → DOOR → NUM,prep:DOOR

where

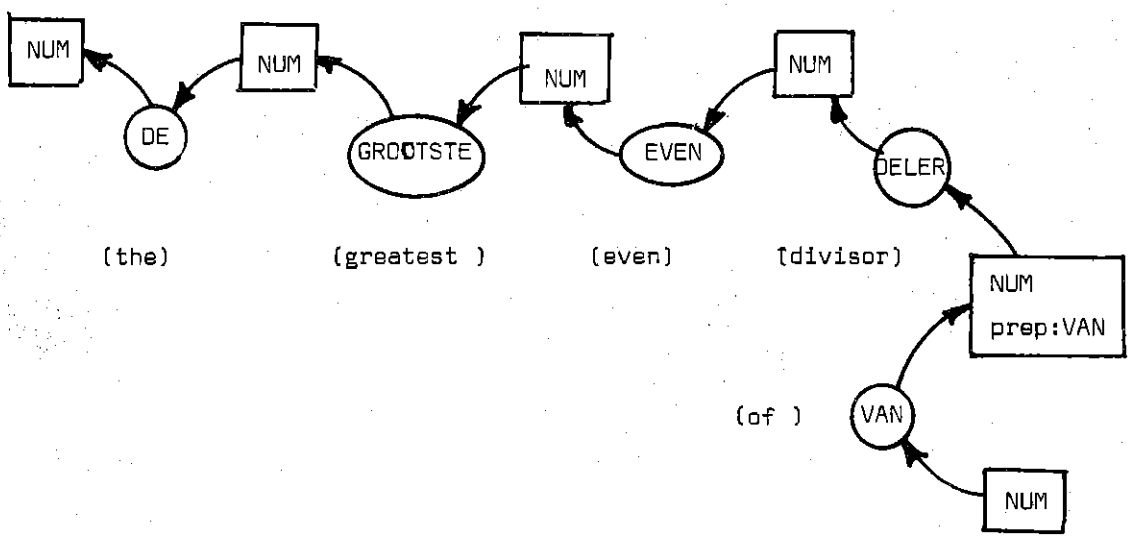
X_1 = {DE, HET, EEN, GROOTSTE, KLEINSTE, EVEN, ONEVEN, ENKELE, ?}
X_2 = {SOM, VERSCHIL, PRODUCT}
X_3 = {DELING}
X_4 = {DELER(S), VIERKANTSWORTEL, TWEEDEMACHT}

(Note that the grammar is clearly not meant for production. A refinement of the argument types should be introduced to rule out certain possibilities. For analysis however the grammar is all right.)

An example of a derivation:

NUM ⇒ DE NUM ⇒ DE GROOTSTE NUM ⇒ DE GROOTSTE EVEN NUM
⇒ DE GROOTSTE EVEN DELER (NUM,prep:VAN) ⇒ DE GROOTSTE EVEN DELER VAN NUM

The corresponding relation structure:



The procedures for the predicates are rather obvious. They simply correspond to their arithmetic equivalents and we won't discuss them in full length.

We did some experiments with an L.U.system based on the above mentionned grammar, the arithmetic procedures and the algorithms 1.1. and 1.2.. The procedures are only defined for integers. Real numbers, if they arise during computation are truncated to integers.

Results of our implementations are illustrated by the following computer output. The same conventions hold as for our experiments with the PC-language in prefix notation. We give a 'literal' English translation of the expressions after wards.

```
MCR>RUN STEELS
?
DE SOM VAN 1 EN 1 ?
INPUT  1: DE SOM VAN 1 EN 1 ?
OUT:   2
?
HET VERSCHIL VAN 5 EN 4 ?
INPUT  2: HET VERSCHIL VAN 5 EN 4 ?
OUT:   1
?
HET PRODUCT VAN 9 EN 3 ?
INPUT  3: HET PRODUCT VAN 9 EN 3 ?
OUT:  27
?
DE DELING VAN 27 DOOR 3 ?
INPUT  4: DE DELING VAN 27 DOOR 3 ?
OUT:   9
?
DELERS VAN 16 ?
INPUT  5: DELERS VAN 16 ?
OUT:   1  2  4  8 16
?
EVEN DELERS VAN 16 ?
INPUT  6: EVEN DELERS VAN 16 ?
OUT:   2  4  8 16
?
ONEVEN DELERS VAN 16 ?
INPUT  7: ONEVEN DELERS VAN 16 ?
OUT:   1
?
ONEVEN DELERS VAN 15 ?
INPUT  8: ONEVEN DELERS VAN 15 ?
OUT:   1  3  5 15
?
ENKELE ONEVEN DELERS VAN 15 ?
INPUT  9: ENKELE ONEVEN DELERS VAN 15 ?
OUT:   1  5
?
EEN ONEVEN DELER VAN 15 ?
INPUT 10: EEN ONEVEN DELER VAN 15 ?
OUT:   3
?
DE KLEINSTE ONEVEN DELER VAN 45 ?
INPUT 11: DE KLEINSTE ONEVEN DELER VAN 45 ?
OUT:   1
?
DE GROOTSTE ONEVEN DELER VAN 45 ?
INPUT 12: DE GROOTSTE ONEVEN DELER VAN 45 ?
OUT:  45
?
ENKELE ONEVEN DELERS VAN 45 ?
INPUT 13: ENKELE ONEVEN DELERS VAN 45 ?
OUT:   1  5 15
```

```
DE VIERKANTSWORTEL VAN 16 ?
INPUT 14: DE VIERKANTSWORTEL VAN 16 ?
OUT:   4
?
DE TWEEDEMACHTSWORTEL VAN 4 ?
UNKNOWN WORD, INPUT NOT ACCEPTED
?
DE TWEEDEMACHT VAN 4 ?
INPUT 15: DE TWEEDEMACHT VAN 4 ?
OUT:  16
?
DE SOM VAN DE KLEINSTE EVEN DELER VAN 16 EN 2 ?
INPUT 16: DE SOM VAN DE KLEINSTE EVEN DELER VAN 16 EN 2 ?
OUT:   4
?
DE GROOTSTE DELER VAN DE VIERKANTSWORTEL VAN 81 ?
INPUT 17: DE GROOTSTE DELER VAN DE VIERKANTSWORTEL VAN 81 ?
OUT:   9
?
DE SOM VAN HET VERSCHIL VAN HET PRODUCT VAN 3 EN 4 EN 3 EN 3 ?
INPUT 18: DE SOM VAN HET VERSCHIL VAN HET PRODUCT VAN 3 EN 4
          EN 3 EN 3 ?
OUT:  12
?
+N\STRUCTURES
?
DE VIERKANTSWORTEL VAN DE SOM VAN 2 EN 2 ?
INPUT 19: DE VIERKANTSWORTEL VAN DE SOM VAN 2 EN 2 ?
OUT:   2
STRUCTURES :
-----------
NODES :
 1.    1  1  1  1  1
 2.   18  7  2 13 18
 3.    4  2  3  1  4
 4.    1  1  4  1  1
 5.    7  5  5  3  7
 6.    4  2  6  1  4
 7.   20  6  4  2 33
 8.   40  5  4  2  9
 9.    6  4  7  1  6
10.   20  9  4  3 34
11.   41  5  5  3 21
12.   20  4  4  2  6
13.   20  3  4  2 34
14.   40  2  4  2  0
15.   20  1  4  2  0
16.   20  0  0  2 21
17.   21  1  8 15 21
RELATIONS :
 1.    2  0 16 15
 2.    2  0 15 14
 3.    2  0 14 13
 4.    2  0 13 12
 5.    3  0 12  8 11
 6.    2  0  8  7
 7.    2  0 11 10
 8.    2  0 16 16
T2 :  16
?
+NO STRUCTURES
?
+END INPUT
```

Translation:

1. The sum of 1 and 1 ?
2. The substraction of 5 and 4 ?
3. The product of 9 and 3 ?
4. The division of 27 by 3 ?
5. Divisors of 16 ?
6. Even divisors of 16 ?
7. Uneven divisors of 16 ?
8. Some uneven divisors of 15?
9. An uneven divisor of 15 ?
10. The smallest uneven divisor of 45 ?
11. The greatest uneven divisor of 45 ?
12. Some uneven divisors of 45 ?
13. The square root of 45 ?
14. The powersquare root of 4 ?
15. The second power of 4 ?
16. The sum of the smallest even divisor of 16 and 2 ?
17. The greatest even divisor of the square root of 81 ?
18. The sum of the difference of the product of 3 and 4 and 3 and 3 ?
19. The square root of the sum of 2 and 2 ?

(+ illustration of the structures switch)

1.6. Some remarks on the distinction between closed completion grammars and phrase structure grammars

Intuitively there is a relation between context-free grammars and closed completion grammars. Indeed, if we have a closed completion grammar $G = (Voa, Vha, Vp, \delta)$ then we can turn it into a cfg. by considering all hidden arguments as nonterminals and all procedure names and occurred arguments as terminals.
If we have a rule $a_1 \ldots a_n \longrightarrow A \longrightarrow a$ then the equivalent one in a cfg. would be $a \longrightarrow A\ a_1 \ldots a_n$.
And clearly it is not too difficult to prove that the languages generated by closed completion grammars are contained in the class of context-free languages.

However the following distinctions can be recognized:

(i) In a cfg. framework we deal with strings, not combinations

(ii) The theoretical status of the hidden arguments is distinct from the one of nonterminals

(iii) In a 'phrase structure' we can express a precedence relation and a dominance relation. In a 'relation structure' we can express a (preferential) precedence relation and a functional relation.

To conclude the distinction between closed comp grammars and cf.grammars lies in the strong generative capacity rather than the weak generative capacity. There remain of course a lot of theoretical problems and we hope to investigate them in the near future.

2. OPEN COMPLETION GRAMMARS

2.1. *Basic definitions*

Now we turn to another type of system generating combinations and assigning relation structures to these combinations, namely an open completion grammar.

Definition 2.1. An *open completion grammar* is a construct $G = (Voa, Vp, \delta)$ where Voa is a finite nonempty set of arguments called the set of occurred arguments, and Vp is a finite nonempty set of procedure names where $Vp \cap Voa = \emptyset$. δ is a finite set each element of which is a finite ternary relation included in $Voa^* \times Vp \times Voa$, relating arguments to procedures.
If $(\sigma, A, a) \in \delta$ where $\sigma \in Va^*$, $A \in Vp$ and $a \in Voa$ then we write $\sigma \rightarrow A \rightarrow a$.
If $(\sigma, A, a_1) \in \delta$ where $\sigma \in Va^*$ and $\sigma = a_1, \ldots a_n$ and $A \in Vp$ then the argument appearing on the right of the rule (the output argument) a_1 is equal to the first argument appearing on the left of the rule. For this reason we also write $\overbrace{a_1 \ldots a_n}^{\downarrow} \rightarrow A$

So the difference between closed and open completion grammars is that the output argument in the second type of systems has already appeared (or is to appear) in the structure, whereas in the first type the output is always an element that must be added to the structure.

Example 2.1.
Let $G = (Voa, Vp, \delta)$ be an open completion grammar and $Voa = \{a,b,c,d\}$ $Vp = \{A, B, C\}$ and δ :

1. a b c → A → a
2. d c b → B → d
3. b a → C → b

An open completion grammar G describes a language called L(G) in the following way. Let R be the set of arguments that appear as output of a procedure ($R \subseteq Voa$) then starting with an arbitrary element of R, put the procedure name of which this argument is output after this element and add all other input arguments to the combination. If there is an argument in the combination that is in R, either the combination is considered complete, or the same method is applied. More formal:

Defintion 2.2. Let $\Rightarrow$ denote the relation is *'preferentially directly derived from'* If there is a combination $x\ a_1\ y$ (x,y possibly empty) where $x,y \in (Voa\ Vp)^*$ and $a_1 \in R$ and if there is a rule in the grammar $a_1 \ldots a_n \rightarrow A \rightarrow a_1$ $(n \geq 1)$ where $a_1, \ldots, a_n \in Voa$ and $A \in Vp$, then we say $x\ a_1\ y \Rightarrow x\ a_1\ A\ a_2 \ldots a_n\ y$.
(Note: when n = 1, with a rule of the form $a_1 \rightarrow A \rightarrow a_1$, then $x\ a_1\ y \Rightarrow x\ a_1 A\ y$)

Also $\overset{*}{\Rightarrow}$ is the reflexive transitive closure of $\Rightarrow$ and $\overset{*}{\Rightarrow}$ will be called *'preferentially derived from'*.

The language generated by an open completion grammar G, called L(G) is defined as $L(G) = \{ x \mid x \in Voa^{\star} \text{ and } y \overset{\star}{=} x \text{ where } y \in R \}$

Example 2.2.

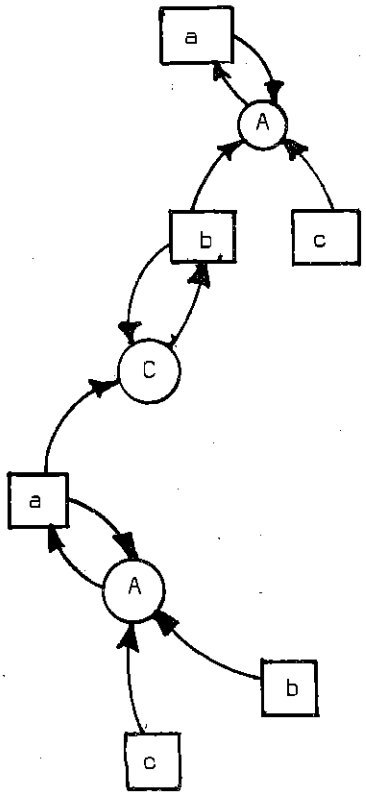
Let G be the completion grammar of example 2.1. then the following derivations are possible:

(i) $a \overset{1}{\Rightarrow} a\ A\ b\ c \overset{3}{\Rightarrow} a\ A\ b\ C\ a\ c \overset{1}{\Rightarrow} a\ A\ b\ C\ a\ A\ b\ c\ c$

(ii) $d \overset{2}{\Rightarrow} d\ B\ c\ b \overset{3}{\Rightarrow} d\ B\ c\ b\ C\ a$

During the derivation process relation structures are obtained in the following way:

Given an occurred argument as output , draw a square for it, connect it with an input AND output relation to the procedure circle and for all input arguments draw squares and make a connection to the procedure circle. For the derivation (i) in example 2.2. this would result in the following structure:



Note that from this example it is very clear that relation structures are graphs and not trees.

To ease our discussion we introduce the following additional terms:

Definition 2.3. Procedures of which the output is an hidden argument will be called *nondepending* procedures. Procedures which are not nondepending will be called *depending*.

The procedures for closed completion grammars are clearly all nondepending while in open completion grammars all procedures are depending.

The remarks we made about preferentiality for closed completion grammars also hold here. The definition of the associated language of an open completion grammar is left to the reader.

Now we turn to the parsing problem for open completion grammars. Again we only treat deterministic open completion grammars due to space limitations.

2.2. *The parsing problem for open completion grammars*

Algorithm 2.1. Let there be a pds. T1 where procedures are stored, a pds.T2 where arguments found in the input but not yet connected in the graph are stored and a pds T3 for all arguments found in the input and connected in the graph. A graphic representation is used for the relation structure.

Let σ be a input combination and σ_i the i-th element in the combination.

Scan the input from left to right.

(a) if σ_i is a procedure

1. create a procedure circle in the structure and put the procedure on T1.
2. check whether there are any arguments on T2 (or on T3 for the first input argument) which can be input to the procedure. If so connect with input relations and (for the first argument) also with an output relation, and put the argument on T3.
   If all arguments are found, that is if the procedure is complete, remove the procedure from T1, and if the output of the procedure is not yet connected to another procedure, put it on T2 and execute the (b) 2 part of this algorithm.

(b) if σ_i is an argument:

1. Create a point in the structure
2. Check for all procedures on T1 whether this argument can be input to it. If so, connect and put it on T3, else put the argument on T2.If the procedure is complete, do the same as under (a) 2. for complete procedures.

To have a grammatical input expression, T1 should be empty, T2 should contain one and only one element (the starting point in the derivation) and the rest should be on T3.

Example 2.3.

Let us take the grammar G of example 2.1. and parse some combinations of L(G).

derivation 1.:

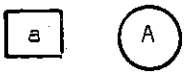
$a \overset{1}{\Rightarrow} a\ A\ b\ c \overset{3}{\Rightarrow} a\ A\ b\ C\ a\ c$

σ = a A b C a c

(i) σ_1 = a

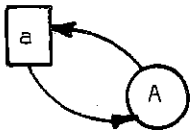
T2: a (argument on T2, create point in structure)

(ii) σ_2 = A



T1: A
T2: a

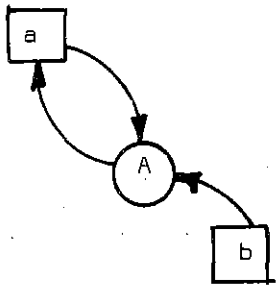
(create procedure circle and put the element on T1



T1: A
T2: -
T3: a

(output was on T2, so connect and put on T3)

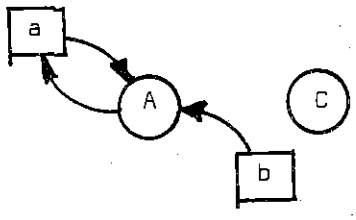
(iii) σ_3 = b



T1: A
T2: -
T3: b a

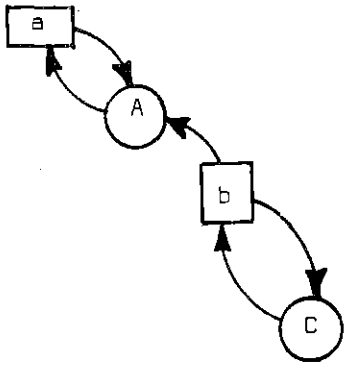
(new input element is argument of procedure

(iv) σ_4 = C



T1: C A
T2: -
T3: b a

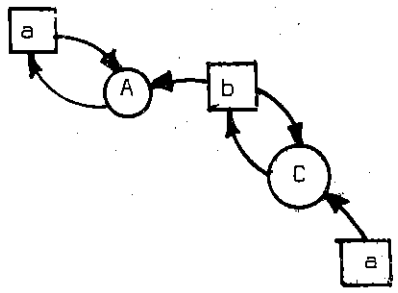
(C as new procedure on T1 and in the structure)



T1: C A
T2: -
T3: b a

b is the output/input of C

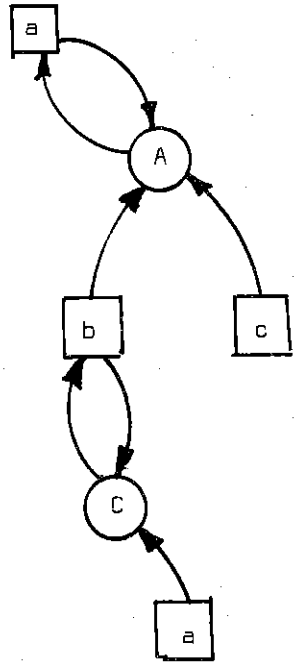
(v) σ_5 = a



T1: C A
T2: -
T3: a b a (a is input for C)

C is complete therefore: T1: A
T2: -
T3: a b a

(vi) σ_6 = c



(č is input for A, hence A is complete)

T1: -
T2: a
T3: c a b

derivation 2:

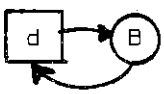
$d \overset{2}{\Rightarrow} d\ B\ c\ b \overset{3}{\Rightarrow} d\ B\ c\ b\ C\ a$

$\sigma = d\ B\ c\ b\ C\ a$

(i) σ_1 = d

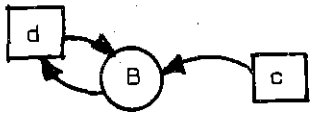
T2: d

(ii) σ_2 = B



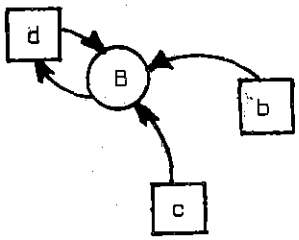
T1: B
T2: -
T3: d

(iii) σ_3 = c



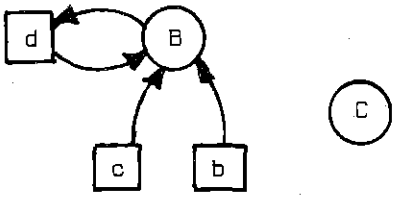
T1: B
T2: -
T3: c d

(iv) σ_4 = b



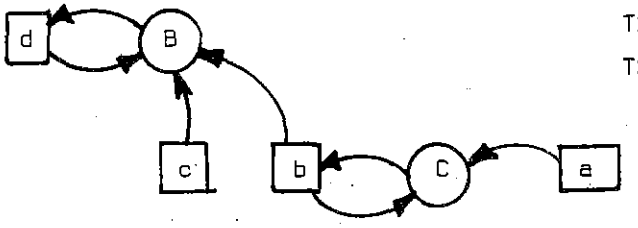
T1: -
T2: d
T3: b c

(v) σ_5 = C



T1: C
T2: d
T3: b c

(vi) σ_6 = a



T1: -
T2: d
T3: a b c

The same remarks on preferentiality of order should be made here as for algorithm 1.1.. Also non preferentially ordered input combinations are to be accepted by the system. As an illustration of this we parse the reverse of derivation 1: recall that σ = a A b C a c now σ = c a C b A

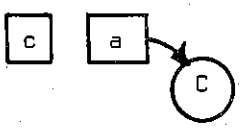
(i) σ_1 = c

T2: c

(ii) σ_2 = a

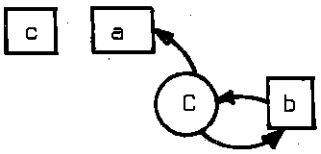
T2: a c

(iii) σ_3 = C



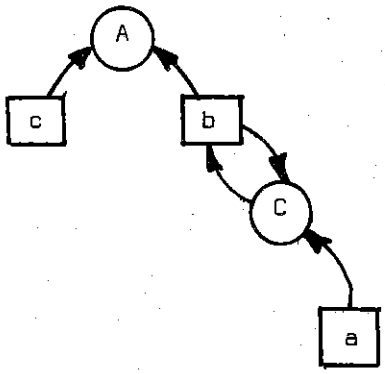
T1: C
T2: c
T3: a

(iv) σ_4 = b



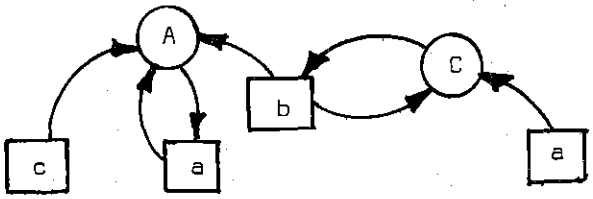
T1: -
T2: b c
T3: a

(v) σ_5 = A



T1: A
T2: -
T3: c b a

(vi) σ_6 = a



T1: -
T2: a
T3: c b a

PROBLEMS 3

(i) Construct a program for algorithm 2.1. in an available programming language and test the examples given.

(ii) Let G = $\langle$ Voa, Vp, δ $\rangle$ be an open completion grammar where Voa = $\{a,b,c,d\}$, Vp = $\{A, B, C\}$ and δ :

1. a b → A → a
2. b c d → B → b
3. d → C → d

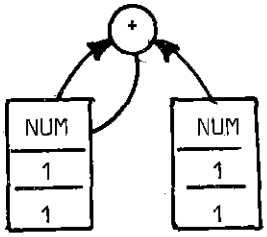
Parse the following examples with algorithm 2.1.

(i) a A b B c d C (ii) C d c B b A a (iii) A a

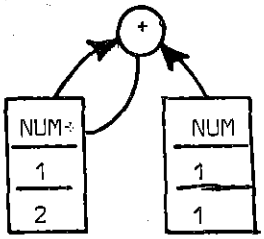
Now we discuss the way in which instant or delayed interpretation influences the parsing process.
Recall from section 1.4. where we discussed the interpretation problem for closed completion grammars that it is possible to define (at least) two interpretation modes: instant or delayed. Suppose now that all procedures in an open completion grammar are considered as instant, then we obtain a situation where it is not possible anymore to use arguments that have been input to some procedure again in another procedure, because if an argument is used, it should be removed from T2 or T3.
Consider e.g. the expression '1 + 1 x 2'. If we leave out all priority rules among the arithmetic procedures, rules which do not count in natural language anyway, then we can have two ways of interpreting '1 + 1 x 2':
(a) (1 + 1) x 2 and (b) 1 + (1 x 2)

The first interpretation is obtained by an instant interpretation mechanism:

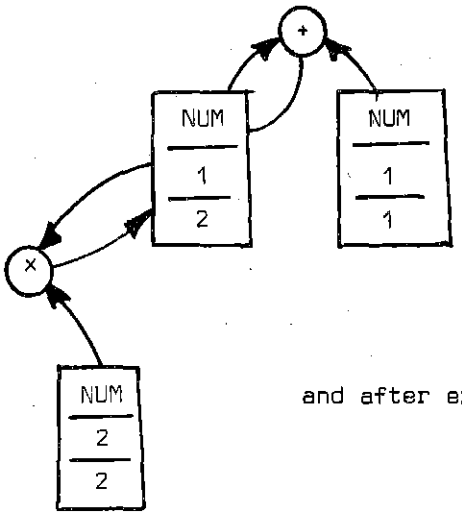
step 1:



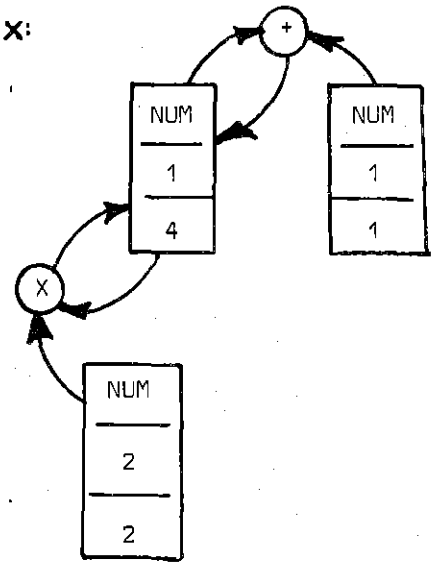
after execution:
of +



step 2:

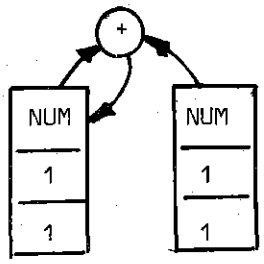


and after execution of X:

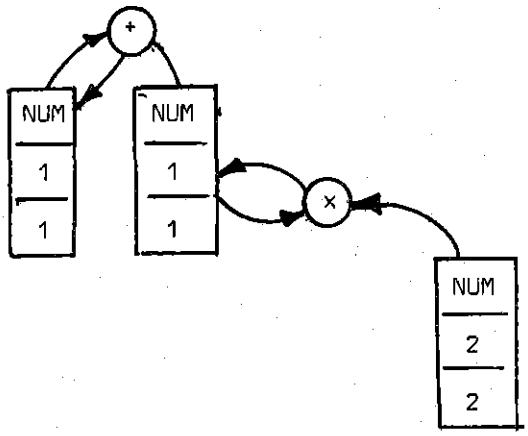


The second interpretation is obtained by a delayed interpretation mechanism. The second argument remains on T3 and is thus open fur further connections.

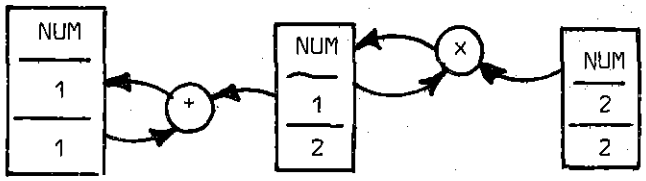
Step:1:



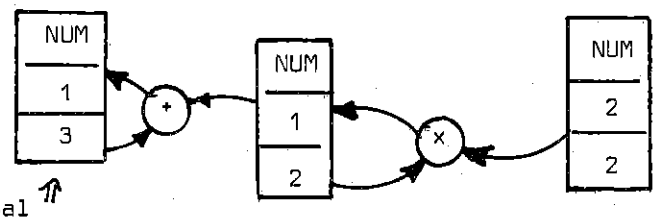
step 2:



After execution of x



After execution of +



final result

This is a very nice illustration of how the way in which interpretation is organized does influence the result of interpretation. Instead of saying this expression is ambiguous, so the grammar must assign more than one structure to it, we say there are different ways of organizing the understanding process and according to the process, we obtain different structures, with the same rule of the grammar.

An interesting point is also that we can define a more economical parsing algorithm if we consider all procedures in an open completion grammar as instant. The algorithm is the following one:

Algorithm 2.2.

Let there be a pds. T1 where all procedures are stored and a pds. T2 for the arguments. We use again a graphic representation for the(partial) relation structure.

Let σ be a given input combination and σ_i the i-th element in the combination

Scan the input from left to right.

A. If σ_i is a procedure:

1. Create a procedure circle in the structure and put the procedure on T1
2. a. Check whether there are any arguments on T2 which can be input to the procedure according to the grammar, if so, connect and take that particular argument from the pds. T2.

   b. if all arguments are found, that is if the procedure is complete, remove the procedures from T1, and execute the B.2. part of this algorithm with as argument the output argument of the procedure.

B. if σ_i is an argument:

1. Create an argument square in the structure
2. Check for all procedures on T1 whether this argument can be input to it. If so connect, else put it on T2. If the procedure is complete, do the same as was specified under A.2.b part of this algorithm.

Just as for closed completion grammars we will now apply the concept of an open completion grammar to the PC-language, this time however preferentially in infix-notation.

2.3. *Application to the PC-language*

In section 1.3. we showed that the PC-language in prefix notation could be treated with closed completion grammars. What we do now is simply change all procedures from nondepending into depending procedures and what we obtain is an open completion grammar generating expressions in infix notation.

Let $G = (Voa, Vp, \delta)$ be an open completion grammar where $Voa = \{LOG\}$ and $Vp = \{NOT, AND, OR, IMPLIES, EQUIVAL, SET, ?\}$ and δ contains the following patterns:

1. LOG LOG $\rightarrow X_1 \rightarrow$ LOG
2. LOG $\rightarrow X_2 \rightarrow$ LOG

where X_1 = {AND, OR, IMPLIES, EQUIVAL, SET}
X_2 = {NOT, ?}

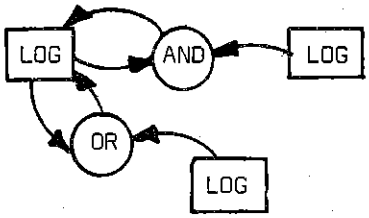
Note that again LOG is the argument type. The argument name can be anything, e.g. TRUE, FALSE, P, Q, and the argument value is assigned by the set-function or fixed.

Some derivations
(i) LOG ⇒ LOG AND LOG ⇒ LOG AND LOG OR LOG
(ii) LOG ⇒ LOG ?

Depending on whether we consider the procedures as instant or delayed we obtain the following structures for derivation (i):

(i) delayed:

(ii) instant:



(Note that NOT comes preferentially after the argument it is negating and not in front of it, NOT seems therefore a procedure which is considered as nondepending even if we have an infix notation)

There are a number of features of an instant interpretation process that makes it more interesting than a delayed one. One of them is that there is less storage required, because once a piece is interpreted, it does not need to be remembered anymore. Also intuitively humans tend to interpret as they go along and not when a whole expression has been produced.

There is however one deficiency, namely that nesting to the right is not possible There is a remedy for this namely the punctuation acting either as a means to turn an instant procedure into a delayed one and vice-versa, either as a means to prevent an argument from being connected to a procedure.
This last solution seems to be present in the case of the PC-language with the use of brackets. Take e.g. P AND (Q OR P) . The first bracket prevents Q from being connected to the AND-procedure. As a result Q remains on T2 and is ready to act as input for the next procedure. The last bracket is breaking up this prevention and the result of Q OR P (stored in the Q-place) is input to the AND-procedure.

Let us now do some experiments again. We programmed algorithm 2.2. in FORTRAN IV and together with the interpretation mechanisms used earlier, the open completion grammar for infix notation and the same procedures as for prefix notation we have a complete L.U. system.

Results of our implementation onthe PDP 11/45 are illustrated by the following output. The same conventions for communicating with the system hold as for previous experiments.

```
?
P SET TRUE
INPUT  1: P SET TRUE
?
Q SET FALSE
INPUT  2: Q SET FALSE
?
P ?
INPUT  3: P ?
OUT: TRUE
?
P AND Q ?
INPUT  4: P AND Q ?
OUT: FALSE
?
P AND Q OR P ?
INPUT  5: P AND Q OR P ?
OUT: TRUE
?
P AND ( Q OR P ) ?
INPUT  6: P AND ( Q OR P ) ?
OUT: TRUE
?
R SET NOT FALSE
INPUT  7: R SET NOT FALSE
?
R ?
INPUT  8: R ?
OUT: TRUE
?
( P IMPLIES Q ) EQUIVAL ( Q IM\MPLIES P ) ?
INPUT  9: ( P IMPLIES Q ) EQUIVAL ( Q IMPLIES P ) ?
OUT: FALSE
?
( ( P AND Q ) OR ( Q AND P ) ) IMPLIES P ?
INPUT 10: ( ( P AND Q ) OR ( Q AND P ) ) IMPLIES P ?
OUT: TRUE
?
+STRUCTURES
?
P EQUIVAL P
INPUT 11: P EQUIVAL P
STRUCTURES :
-----------
NODES :
 1.   10  0  0  0 14
 2.    5  5  1  0  5
 3.   10  2  5  0 14
RELATIONS :
 1.    3  0  1  1  3
T2 :   1
?
+NO STRUCTURES
?
```

```
+GRAMMAR
 1.    3 10 10 10
 2.    3 10 10 10
 3.    3 10 10 10
 4.    2 10 10
 5.    3 10 10 10
 6.    2 10 10
 7.    2 10 10
 8.    3 10 10 10
 9.    3 10 10 11
?
+LEXICON
 1. AND        1  1  1  0
 2. OR         2  2  1  0
 3. IMPLIES    3  3  1  0
 4. NOT        4  4  1  0
 5. EQUIVAL    5  5  1  0
 6. ?          6  6  1  0
 7. PROOF      7  7  1  0
 8. SET        8  8  1  0
 9. (          9  9  0  0
10. LOG       10  0 10  2
11  TRUE      10  0 11  0
12. FALSE     10  0 12  1
13. )         11  0 13  0
14. P  10  0 14  0
15. Q  10  0 15  1
16. R  10  0 16  0
?
S SET NOT Q
INPUT 12: S SET NOT Q
?
Q ?
INPUT 13: Q ?
OUT: FALSE
?
Q NOT ?
INPUT 14: Q NOT ?
OUT: TRUE
?
AND P Q ?                  From input 15 we start to experiment
INPUT 15: AND P Q ?        with non preferential orders, in particular
OUT: FALSE                 prefix and postfix.
?
IMPLIES P Q ?
INPUT 16: IMPLIES P Q ?
OUT: FALSE
?
IMPLIES Q P ?
INPUT 17: IMPLIES Q P ?
OUT: TRUE
?
P Q IMPLIES ?
INPUT 18: P Q IMPLIES ?
OUT: FALSE
?
Q P IMPLIES ?
INPUT 19: Q P IMPLIES ?
OUT: TRUE
?
AND OR AND P Q R S ?                Note that input 20 is equal to input 21 and
INPUT 20: AND OR AND P Q R S ?      23, only the preferential order is different.
OUT: TRUE
?
P AND Q OR R AND S ?
INPUT 21: P AND Q OR R AND S ?
OUT: TRUE
?
S R P Q AND OR AND ?
INPUT 22: S R P Q AND OR AND ?
OUT: FALSE
?
S R Q P AND OR AND ?
INPUT 23: S R Q P AND OR AND ?
OUT: TRUE
?
SET INP! TRUE
INPUT 24: SET INP! TRUE
```

```
?
SET INP2 FALSE
INPUT 25: SET INP2 FALSE
?
INP1\! AND INPT\2 ?
INPUT 26: INP! AND INP2 ?
OUT: FALSE
?
+LEXICON
 1. AND        1  1  1  0
 2. OR         2  2  1  0
 3. IMPLIES    3  3  1  0
 4. NOT        4  4  1  0
 5. EQUIVAL    5  5  1  0
 6. ?          6  6  1  0
 7. PROOF      7  7  1  0
 8. SET        8  8  1  0
 9. (          9  9  0  0
10. LOG       10  0 10  2
11. TRUE      10  0 11  0
12. FALSE     10  0 12  1
13. )         11  0 13  0
14. P  10  0 14  0
15. Q  10  0 15  1
16. R  10  0 16  0
17. S  10  0 17  0
18. INP!  10  0 18  0
19. INP2  10  0 19  1
?
+END INPUT
```

INP! en inp2 are two new variable names, this just to illustrate that anything new is considered as a propositional variable

```
MCR>PIP
```

2.4. *Application to natural language*

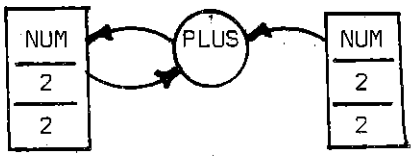
In section 1.5. we showed that nominal groups can be treated with closed completion grammars. In this section we will extend our discussion to other parts of speech which appear outside the nominal phrases. We stress that we do not present a fully worked out theory but only indicate a direction of research. The universe of discourse is again simple arithmetics, and the language is Dutch.

(i) Nouns are the only possible way of expressing arguments. For this purpose we will use them but leave in this section all nouns out which are procedures.

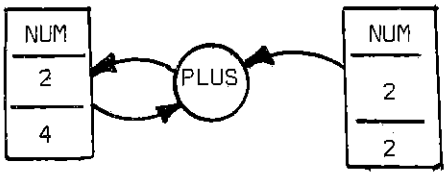
(ii) Prepositions. We have seen in section 1.5. some prepositions which were only used as indicators of a certain relationship. Now we discuss some prepositions which are more than this. E.g. PLUS (plus) MIN (minus), MAAL (times, there is a difference here between English and Dutch,'maal'is a preposition but 'times' isn't)

E.g.: 2 PLUS 2
(2 plus 2)

the relation structure:



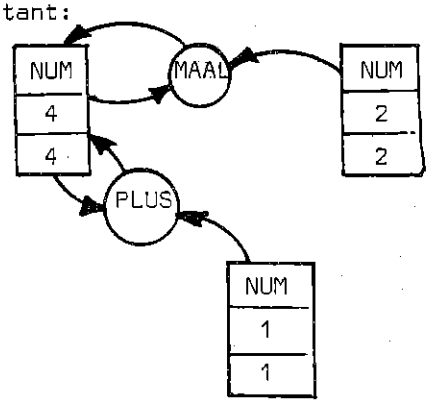
after execution:



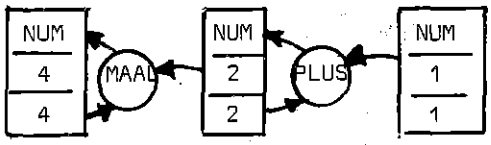
To illustrate the ambiguity and distinction between delayed and instant procedures consider the following example:

4 MAAL 2 PLUS 1
(4 times 2 plus 1)

(i) instant:



(ii) delayed:

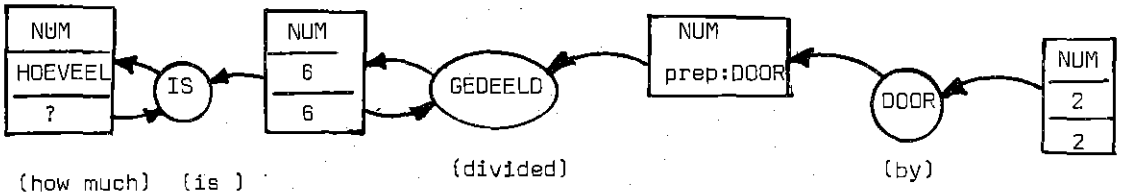


(iii) Verbs: Although the matter needs further investigation verbs seem to be depending procedures. The first input (also output) argument is what is traditionally called the subject of the sentence. This is in accordance with the fact that the subject of a sentence is standing preferentially in front of the sentence and also that subject and main verb agree in number.
When verbs are used in the imperative (and interrogative) they are preferentially in front position. This seems to be because then the output is not present in the input combination but is created as an hidden argument. In other words when verbs are used in imperative or interrogative, they shift from depending into nondepending procedures

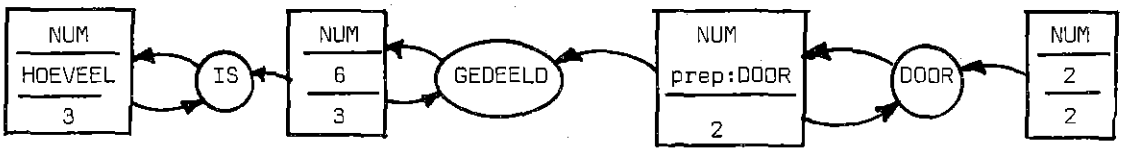
(iv) participles are used in the same way as verbs. Consider e.g.
VERMENIGVULDIGD (multiplied), VERMEERDERD (augmented), VERMINDERD (decreased), GEDEELD (divided),etc..

E.g.: HOEVEEL IS 6 GEDEELD DOOR 2 ?
HOWMUCH IS 6 DIVIDED BY 2 ?

structure:



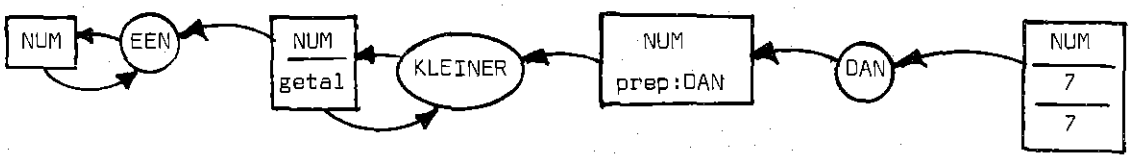
after execution of the functions:



(Note that 'door' is a nondepending procedure)

(v) in an equivalent way adjectives when appearing after a noun (instead of in front of it) are used.

E.g; EEN GETAL KLEINER DAN 7
(a number smaller than 7



There is a lot more to say (e.g. about relative clauses and conjunction) but this will do as an illustration.

The reader may have felt the need for a system in which both depending and nondepending procedures are appearing. We will introduce such a system called a complex completion grammar in the next section. Experiments on natural language processing for the parts of speech that were discussed in this section will be postponed till then.

3. COMPLEX COMPLETION GRAMMARS

3.1. *Basic definitions*

Now we define a 'mixed' type of grammar, which accepts the union of the language accepted by open and closed completion grammars.

Definition 3.1. A *complex completion grammar* is a quintuple $G = \langle Voa, Vha, Vd, Vn, \delta \rangle$ where

Voa is a finite nonempty set of arguments called the set of occurred arguments

Vha is a finite nonempty set of arguments called the set of hidden arguments

$Voa \cup Vha = Va$, is the set of arguments

Vd is a finite set of procedure names called the set of depending procedures

Vn is a finite set of procedure names called the set of nondepending procedures.

$Vd \cup Vn = Vp$, is the set of procedures and $Vp \cap Va = \emptyset$

$\delta \subseteq Va^* \times Vp \times Va$ is a complex function relating arguments to procedure names.

If $\langle \sigma, A, a \rangle \in \delta$ then we write $\sigma \rightarrow A \rightarrow a$ where $\sigma \in Va^*$, $A \in Vp$ and $a \in Va$.

Definition 3.2. Let $\Rightarrow$ denote the relation *'is preferentially directly derived from'*

- If there is a combination $x\ u\ y$ (x,y possibly empty) where $x,y \in (Va \cup Vp)^*$ $u \in Vha$ and if there is a rule in the grammar of the form $a_1 \ldots a_n \rightarrow A \rightarrow a$ $(n \geq 1)$ where $a_1, \ldots, a_n \in Va$ and $A \in Vn$, then we say that $x\ u\ y \Rightarrow x\ A\ a_1 \ldots a_n y$.

- Or if there is combination $x\ u\ y$ (x,y, possibly empty) where $x,y \in (Va \cup Vp)^*$, $u \in Voa$ and if there is a rule in the grammar $ua_1 \ldots a_n \rightarrow A \rightarrow u$ where $a_1, \ldots, a_n \in Va$ and $A \in Vd$, then we say that $x\ u\ y \Rightarrow x\ u\ A\ a_1 \ldots a_n\ y$

$\overset{*}{\Rightarrow}$ is the reflexive transitive closure of $\Rightarrow$ and we call $\overset{*}{\Rightarrow}$ *'is preferentially derived from'*.

The language generated by a complex completion grammar G, denoted as L(G) is defined as

$L(G) = \{ x \mid x \in (Voa \cup Vp)^* \text{ and } y \overset{*}{\Rightarrow} x \text{ where } y \in Va \}$

Example 3.1. Let $G = \langle Voa, Vha, Vd, Vn\ \delta \rangle$ be a complex completion grammar where $Voa = \{a,b,c,d\}$ $Vha = \{a\}$, $Vd = \{A,B\}$ $Vn = \{C,D\}$ and δ

1. $d\ b \rightarrow A \rightarrow d$
2. $c\ a \rightarrow C \rightarrow a$
3. $b\ d \rightarrow D \rightarrow a$
4. $c \rightarrow B \rightarrow c$

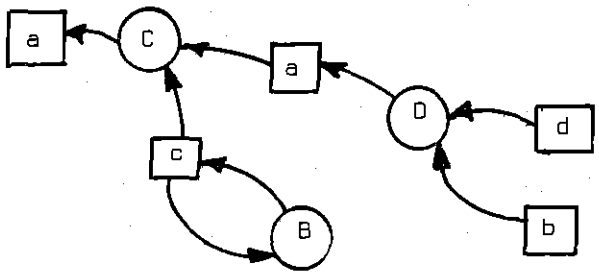
Some derivations:

(i) $a \overset{2}{\Rightarrow} C\ c\ a \overset{4}{\Rightarrow} C\ c\ B\ a \overset{3}{\Rightarrow} C\ c\ B\ D\ b\ d$

(ii) $d \overset{1}{\Rightarrow} d\ A\ b$

RElation structures are obtained in the same way as for open and closed completion grammars. I.e. when there is a nondepending procedure, connect the output with only one line, whereas if the procedure is depending, connect the o-utput with an input and output relation.

The relation structure for derivation (i) is:



3.2. *The parsing problem for complex completion grammars*

The algorithm that solves the problem is basically a composition of algorithm 1.1 and algorithm 2.2..

Algorithm 3.1.

Let there be a pds. T1 for the procedures and a pds. T2 for the arguments.
Let σ be a given input combination and σ_i the i-th element in the combination.

Scan the input from left to right.

A. If σ_i is a nondepending procedure.

1. create a procedure circle in the structure and put the procedure on T1.
2. (a) Check whether there are any arguments on T2 which can be input to the procedure according to the grammar, if so connect and take that particular argument from the pds. T2.

   (b) If all arguments are found, that is if the procedure is complete, remove the procedures from T1, put the output element as argument square in the structure and connect it with an output relation to the procedure, then execute the C.2. part of this algorithm.

B. If σ_i is a depending procedure:

1. create a procedure circle in the structure and put the procedure on T1.
2. (a) Check whether there are any arguments on T2 which can be input to the procedure according to the grammar, if so, connect and take that particular argument from the pds.T2. Note that we connect with input and output relations if it is the first input argument.

   (b) If all arguments are found, that is if the procedure is complete, remove the procedure from T1 and execute the C.2. part of this algorithm with as argument the output argument of the procedure.

C. If ξ_i is an argument:

1. Create an argument square in the structure
2. Check for all procedures on T1 whether this argument can be input to it. If so connect, else put it on T2. If the procedure is complete and depending, execute the B.2. part of this algorithm. If the procedure is complete and nondepending, execute the A.2.b, part of this algorithm.

The reader is advised to work out some examples himself. He will see that the parsing process is identical for depending procedures with the one introduced by algorithm 2.2. and for nondepending with the one introduced by algorithm 1.1..

3.3. *Application to natural language*

We promised in section 2.4. to do experiments with a grammar containing depending as well as nondepending procedures in a natural language environment.

The grammar is the following one:

Let G = ⟨Voa, Vha, Vd, Vn, δ ⟩ be a complex completion grammar and

Voa = {HOEVEEL, WAT} and all natural numbers

Vha = {NUM, (Num,prep:VAN) , (Num, prep:MET), (NUM, prep:DOOR), (Num,PREP:en)}

Vd = {IS, VERMENIGVULDIGD, GEDEELD, VERMINDERD, VERMEERDERD, PLUS, MAAL, MIN}

Vn = {DE, HET, EEN, VAN, DOOR, EN, SOM, VERSCHIL, PRODUCT, DELING, DELER, DELERS, GROOTSTE, KLEINSTE, EVEN, ONEVEN, ENKELE, VIERKAN$WORTEL, TWEEDEMACHT, MET}

δ contains the following patterns:

1. NUM → X_1 → NUM
2. NUM → VAN → (NUM, prep/VAN)
3. NUM → EN → (NUM,prep:EN)
4. NUM → DOOR → (NUM,prep:DOOR)
5. NUM → MET → (NUM, prep:MET)
6. (NUM,prep:VAN) (NUM,prep:EN) → X_2 → NUM
7. (NUM.prep:VAN) (NUM,prep:DOOR) → DELING → NUM
8. NUM NUM → X_4 → NUM
9. NUM (NUM,prep:MET) → X_5 → NUM
10. NUM (NUM,prep:DOOR) → GEDEELD → NUM

We made an implementation with this grammar, the interpretation mechanism and algorithm 1.2.. The procedures should be rather obvious here . Note that e.g. VERMINDERD (diminished), VERSCHIL (substraction), MIN (minus), make all use of the same procedural definition.
Our usual conventions hold for communicating with the system.
An ENglish translation will be given afterwards.

```
?
HOEVEEL IS 1 PLUS 1 ?
INPUT  1: HOEVEEL IS 1 PLUS 1 ?
OUT:   2
?
HOEVEEL IS 3 MIN 2 ?
INPUT  2: HOEVEEL IS 3 MIN 2 ?
OUT:   1
?
HOEVEEL IS 3 MAAL 2 ?
INPUT  3: HOEVEEL IS 3 MAAL 2 ?
OUT:   6
?
HOEVEEL IS 6 GEDEELD DOOR 3 ?
INPUT  4: HOEVEEL IS 6 GEDEELD DOOR 3 ?
OUT:   2
?
HOEVEEL IS 7 VERMINDERD MET 3 >
UNKNOWN WORD, INPUT NOT ACCEPTED
?
HOEVEEL IS 7 VERMINDERD MET 3 ?
INPUT  5: HOEVEEL IS 7 VERMINDERD MET 3 ?
OUT:   4
?
WAT IS DE SOM VAN 4 EN 5 GEDEELD DOOR 3 ?
INPUT  6: WAT IS DE SOM VAN 4 EN 5 GEDEELD DOOR 3 ?
OUT:   3
?
HOEVEEL IS 7 GEDEELD DOOR DE SOM VAN 4 EN 3 ?
INPUT  7: HOEVEEL IS 7 GEDEELD DOOR DE SOM VAN 4 EN 3 ?
OUT:   1
?
HOEVEEL IS DE VIERKANTSWORTEL VAN 16 MAAL 4 ?
INPUT  8: HOEVEEL IS DE VIERKANTSWORTEL VAN 16 MAAL 4 ?
OUT:  16
?
VAN 1 EN 1 DE SOM IS HOEVEEL ?
INPUT  9: VAN 1 EN 1 DE SOM IS HOEVEEL ?     Note the non prefe rential input
OUT:   2
?
7 MAAL 7 GEDEELD DOOR 7 MIN 1 PLUS 7 IS HOEVEEL ?
INPUT 10: 7 MAAL 7 GEDEELD DOOR 7 MIN 1 PLUS 7 IS HOEVEEL ?

OUT:  13
?
DE GROOTSTE ONEVEN DELER VAN DE VIERKANTSWORTEL VAN 81 PLUS 1 ?
INPUT 11: DE GROOTSTE ONEVEN DELER VAN DE VIERKANTSWORTEL VA
 81 PLUS 1 ?
OUT:  10
?                    INput 12 is not accepted because the character 'O'
10 MAAL 10 ?
INPUT 12: 1O         was given instead of the number 'Ø'.
?
10 MAAL 10 ?
INPUT 13: 10 MAAL 10 ?
OUT: 100
?
100 GEDEELD DOOR DE TWEEDEMACHT VAN 10 ?
INPUT 14: 100 GEDEELD DOOR DE TWEEDEMACHT VAN 10 ?
OUT:   1
?
DE KLEINSTE ONEVEN DELER VAN 100 VERMENGI\NIGVULDIGD MET 15 ?
INPUT 15: DE KLEINSTE ONEVEN DELER VAN 100 VERMENIGVULDIGD M
T 15 ?
OUT:  15
```

```
?
HOEVEEL IS 17 GEDEELD DOOR 17 VERMENIGVULDIGD MET 2 GEDEELD DOOR 2 ?
INPUT 16: HOEVEEL IS 17 GEDEELD DOOR 17 VERMENIGVULDIGD MET
 GEDEELD DOOR 2 ?
OUT:   1
STRUCTURES :
-----------
NODES :
 1.   20  0  0  2 22
 2.   24  8  1 16 24
 3.   20  2  5  2 33
 4.   26 10  2  6 26
 5.    5  3  3  1  5
 6.   20  5  4  3 34
 7.   42  4  5  3 34
 8.   25 11  4  5 25
 9.   29  9  5  1 29
10.   20  9  4  4 35
11.   43  8  5  4 37
12.   26 10  6  6 26
13.    5  3  7  1  5
14.   20 13  4  5 36
15.   42 12  5  5 35
16.   21  1  8 15 21
RELATIONS :
 1.    3  0  1  1  3
 2.    3  0  1  1  7
 3.    2  0  7  6
 4.    3  0  1  1 11
 5.    2  0 11 10
 6.    3  0  1  1 15
 7.    2  0 15 14
 8.    2  0  1  1
T2 :   1
?
+NO STRUCTURES
?
DE VIERKANTSWORTEL VAN DE SOM VAN 4 EN 4 EN 4 ?
INPUT 17: DE VIERKANTSWORTEL VAN DE SOM VAN 4 EN 4 EN 4 ?
OUT:   4
UNGRAMMATICAL INPUT
?
DE VIERKANTSWORTEL IS ?
INPUT 18: DE VIERKANTSWORTEL IS ?
UNGRAMMATICAL INPUT
?
IS WAT DE EEN
INPUT 19: IS WAT DE EEN
UNGRAMMATICAL INPUT
?
VAN 16 DE VIERKANTSWORTEL GEDEELD DOOR 4 ?
INPUT 20: VAN 16 DE VIERKANTSWORTEL GEDEELD DOOR 4 ?
OUT:   1
?
+STRUCTURES
?
VAN 16 DE VIERKANTSWORTEL DOOR 4 GEDEELD ?
INPUT 21: VAN 16 DE VIERKANTSWORTEL DOOR 4 GEDEELD ?
OUT:   1
STRUCTURES :
-----------
NODES :
 1.    4  2  1  1  4
 2.   20  1  4  2 33
 3.   40  5  4  2  1
 4.    1  1  2  1  1
 5.   18  7  3 13 18
 6.   20  4  4  1  4
 7.   20  8  4  1 33
 8.    5  3  4  1  5
 9.   42 11  5  3  5
10.   20  0  0  3 34
11.   26 10  5  6 26
12.   21  1  6 15 21
RELATIONS :
 1.    2  0  3  2
 2.    2  0  7  6
 3.    2  0  3  3
 4.    2  0  7  7
 5.    3  0  9 10  9
 6.    2  0 10 10
T2 :   12
```

From input 20 we start to experiment systematically with non preferentially orderings. They are all being processed as one can see.

```
DOOR 4 GEDEELD DO\E ^U
DOOR 4 GEDEELD DE VE\IERKANTSWORTEL VAN 16 ?
INPUT 23: DOOR 4 GEDEELD DE VIERKANTSWORTEL VAN 16 ?
OUT:   1
STRUCTURES :
-----------
NODES :
 1.    5  3  1  1  5
 2.   20  1  4  2 33
 3.   42  4  5  2  1
 4.   26 10  2  6 26
 5.    1  1  3  1  1
 6.   18  7  4 13 18
 7.    4  2  5  1  4
 8.   20  7  4  3 34
 9.   40  6  4  3  5
10.   20  5  4  3 34
11.   20  0  3  3 26
12.   21  1  6 15 21
RELATIONS :
 1.    2  0  3  2
 2.    3  0 11 11  3
 3.    2  0 11 10
 4.    2  0 10  9
 5.    2  0  9  8
 6.    2  0 11 11
T2 :   11
?
DOOR 4 DE VIERKANTSWORTEL VAN 16 GEDEELD ?
INPUT 24: DOOR 4 DE VIERKANTSWORTEL VAN 16 GEDEELD ?
OUT:   1
STRUCTURES :
-----------
NODES :
 1.    5  3  1  1  5
 2.   20  1  4  2 33
 3.   42 11  5  3  1
 4.    1  1  2  1  1
 5.   18  7  3 13 18
 6.    4  2  4  1  4
 7.   20  6  4  3 34
 8.   40  5  4  3 34
 9.   20  4  4  3  5
10.   20  0  0  3 34
11.   26 10  5  6 26
12.   21  1  6 15 21
RELATIONS :
 1.    2  0  3  2
 2.    2  0 10  9
 3.    2  0  9  8
 4.    2  0  6  7
 5.    3  0  3 10  3
 6.    2  0 10 10
T2 :   10
?
DOOR 4 VAN 16 DE VIERKANTSWORTEL GEDEELD ?
INPUT 25: DOOR 4 VAN 16 DE VIERKANTSWORTEL GEDEELD ?
OUT:   1
STRUCTURES :
-----------
NODES :
 1.    5  3  1  1  5
 2.   20  1  4  2 33
 3.   42 11  5  3  1
 4.    4  2  2  1  4
 5.   20  4  4  3 34
 6.   40  8  4  3  4
 7.    1  1  3  1  1
 8.   18  7  4 13 18
 9.   20  7  4  3  5
10.   20  0  0  3 34
11.   26 10  5  6 26
12.   21  1  6 15 21
RELATIONS :
 1.    2  0  3  2
 2.    2  0  6  5
 3.    2  0 10  9
 4.    2  0  6  6
```

```
 5.    3  0  3 10  3
 6.    2  0 10 10
T2 :   10
?
GEDEELD DOOR 4 VAN 16 DE VIERKANTSWORTEL ?
INPUT 26: GEDEELD DOOR 4 VAN 16 DE VIERKANTSWORTEL ?
OUT:    1
STRUCTURES :
-----------
NODES :
 1.   26 10  1  6 26
 2.    5  3  2  1  5
 3.   20  2  4  2 33
 4.   42  1  5  2  4
 5.    4  2  3  1  4
 6.   20  5  4  3 34
 7.   40  9  4  3  1
 8.    1  1  4  1  1
 9.   18  7  5 13 18
10.   20  8  4  3 34
11.   20  0  3  3 26
12.   21  1  6 15 21
RELATIONS :
 1.    3  0 11 11  4
 2.    2  0  4  3
 3.    2  0  7  6
 4.    2  0 11 10
 5.    2  0  7  7
 6.    2  0 11 11
T2 :   11
?
+LEXICON
 1. DE          1  1  0  1
 2. HET         2  1  0  1
 3. EEN         3  1  0  2
 4. VAN         4  2  0  1
 5. DOOR        5  3  0  1
 6. EN          6  4  0  1
 7. SOM         7  5  0  3
 8. VERSCHIL    8  5  0  4
 9. PRODUCT     9  5  0  5
10. DELING     10  6  0  6
11. DELER      11  7  0  7
12. DELERS     12  7  0  7
13. GROOTSTE   13  1  0  8
14. KLEINSTE   14  1  0  9
15. EVEN       15  1  0 10
16. ONEVEN     16  1  0 11
17. ENKELE     17  1  0 12
18. VIERKANT   18  7  0 13
19. TWEEDEMA   19  7  0 14
20. NUM        20  0 20  1
21. ?          21  1  1 15
22. HOEVEEL    20  0 22  1
23. WAT        20  0 23  1
24. IS         24  8  1 16
25. VERMENIG   25 11  1  5
26. GEDEELD    26 10  1  6
27. VERMINDE   27 11  1  4
28. VERMEERD   28 11  1  3
29. MET        29  9  0  1
30. PLUS       30  8  1  3
31. MIN        31  8  1  4
32. MAAL       32  8  1  5
?
+GRAMMAR
 1.    2 20 20
 2.    2 40 20
 3.    2 42 20
 4.    2 41 20
 5.    3 20 40 41
 6.    3 20 40 42
 7.    2 20 40
 8.    3 20 20 20
 9.    2 43 20
10.    3 20 20 42
11.    3 20 20 43
?
```

Translation:

1. How much is 1 plus 1 ?
2. How much is 3 minus 2 ?
3. How much is 6 divided by 3 ?
4. How much is 7 diminished by 3
5. How much is 7 diminished by 3 ?
6. What is the sum of 4 and 5 divided by 3 ?
7. How much is 7 divided by the sum of 4 and 3 ?
8. How much is the square root of 16 times 4 ?
9. Of 1 and 1 the sum is how much ?
10. 7 times 7 divided by 7 minus 1 plus 7 is how much ?
11. The greatest uneven divisor of the square root of 81 plus 1 ?
12. 10 times 10 ?
13. 1Ø times 1Ø ?
14. 1ØØ divided by the square root of 1Ø ?
15. The smallest uneven divisor of 1ØØ multiplied by 15 ?
16. HOw much is 17 divided by 17 multiplied by 2 divided by 2 ?
    (with structures switch)
17. The square root of the sum of 4 and 4 and 4 ?
18. The square root is ?
19. Is what the an
20. Of 16 the square root divided by 4 ?
21. Of 16 the square root by 4 divided ?
22. Of 16 the square root divided by 4 ?
23. By 4 divided the square rœt of 16 ?
24. By 4 the square root of 16 divided ?
25. By 4 of 16 the square root divided ?
26. Divided by 4 of 16 the square root ?

4. PERSPECTIVES AND CONCLUSIONS

4.1. Perspectives

Although some insights may have been achieved, a lot of problems remain. In particular how world knowledge should be represented and incorporated in the process of parsing.
The following points should be investigated further:

(i) Refinement of the types within an argument. The type information that forms the basis for a connection in the structure during parsing was here presented as being a simple and straightforward matter. This is clearly not the case, also there some preferentiality is involved, as was recognized and worked out by Wilks (1975).

(ii) Lexical ambiguity. The definition of nondeterministic parsing algorithms should be undertaken for the three types of systems. Clearly nondeterminism (as defined in definition 1.6) is equal to a certain type of lexical ambiguity.

(iii) Refinement of the interpretation mechanism. When an understander meets the expression 'Give me the names of some human beings', he cannot start to enumerate all beings, then compute the subset of human beings and finally return a subset of this, simply because the set of beings is an infinite set. What we need therefore is a sort of intertwined interpretation mechanism , where each procedure is not executed seperately.
In particular it should be partially executed till information can be passed to the procedure from which the output of the current procedure is depending. Then this procedure is partially executed and so on. Semantic interpretation in this way runs up and down a structure, preventing excessive computation.

(iv) Procedural definition of predicates. A lot of work remains in discovering what procedures are used for the different predicates. The problem is a difficult one because it hangs together with the way in which the memory (or data base) is organized.
We refer in this context to recent work of Hewitt (1973,1975), Winograd (1975) and others.

(v) Intermediate representations . Another way of solving the problem of excessive computation is by introducing intermediate representations for sets. It may be thought that the procedures can only be direct mappings, i.e. functions themselves, however it is perfectly possible to let the procedures be such things as 'set-builders'. E.g. 'Some numbers smaller than 6'. 'Number' can be considerd as a procedure having as output $\{x \mid \text{Number}(x)\}$ 'smaller' takes this set and turns it into a new form: $\{x \mid \text{NUM}(x) \text{ and } x < 6\}$,etc...
We will deal with these matters in forthcoming publications.

4.2. Conclusions

To conclude we state some of the insights we hope to have made clear.

1. The understanding mechanism is basically a set of processing systems that bring about understanding by the manipulation of information structures. One of them is a parser, that is a system extracting structures according to a given grammar for an arbitrary input. Another is an interpreter, a system carrying out the interpretation of the meaning elements in the structure obtained by the parser. So the parser and interpreter communicate via a structure (called the relation structure in this paper).

Contrary to structural (and in particular Chomskyan) linguists we do not think that structures (structural descriptions on the level of syntax and unordered (or ordered) lists of semantic markers on the level of semantics) are a final and sufficient explanation for understanding.
Instead of studying structures, we should study procedures. Structures are only a by-product of the functioning of the processing procedures.

2. One of the main novelties introduced is the attitude towards order. Order is here not simply a feature of the structure of a language, but is something that can be understood from the way in which the parsing proceeds. In other words, order is not an end in itself, but motivated by the understanding process. It is no coincidence that the subject of the sentence is standing preferentially in front of the verb, that the adjectives and adverbs stand in front of the noun, that prepositions come before every other word in the noun phrase ,etc..
This can all be explained from the role they play in the parsing process.

A very strong result is also the flexibility of the parsing process, something completely lacking from phrase structure parsing.

3. Another interesting point is that semantic interpretation is not taking place w n syntactic processing is finished for the whole sentence. We showed that there are other ways of doing this and also that the interpretation itself is depending on the way in which the process of interpreting is conceived.

4. Other ways of extracting semantic structures without doing first phrase structure parsing are Riesbeck's parser (Riesbeck,1974) producing Schank's conceptual dependency graphs and Wilks' analyzer (Wilks,1975) . Our approach differs from those mentionned above, especially the first one, in that we tried to define underlying systems, instead of just designing a program doing the job.
The need for relationally directed descriptions of language is something also felt more and more felt in structural linguistics (cf. Johnson,1974).

There remains a lot to be discovered and investigated. We personally feel that the systems described have a great potentiality in them.
We hope that completion grammars will turn out to be an interesting tool enlarging our capacity to deal with language.

5. REFERENCES

Hewitt, C. (1973) Procedural Semantics. In: Rustin, R. (ed): Natural language Processing. Algorithmics Press, N.Y..

Hewitt, C. (1975) Stereotypes. In: Schank, R. & B.L. Nash-Weber (ed) Theoretical Issues in Natural language processing. Mimeo Cambr-idge Mass.

Johnson, D.E. (1974) Toward a theory of relationally based grammar. Ph.D. Thesis Urbana, Illinois.

Riesbeck, C.K.(1974) Compuational understanding: analysis of sentences and context. Institute for semantics and cognitive studies. Castagnola, microfiche.

Wilks, Y. (1975) An intelligent analyzer and understander of English. Comm. ACM. 1975

Winograd, T. (1975) Frame representations and the declarative procedural controversy. In D. Bobrow & A. Collins (eds): Representation and understanding. N.Y. Academic Press;