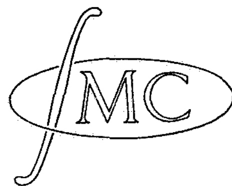
STICHTING
MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49
AMSTERDAM
REKENAFDELING

MR 72

On the construction of ALGOL-procedures for generating, analysing and translating sentences in natural languages

by

C. H. A. Koster



February, 1965

BIBLIOTHEEK MATHEMATISCH CENTRUM
AMSTERDAM

Fl.1,80

Printed at the Mathematical Centre at Amsterdam,49,2nd Boerhaavestraat.
The Netherlands.

The Mathematical Centre, founded the 11th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications, and is sponsored by the Netherlands Government through the Netherlands Organization for Pure Scientific Research (Z.W.O.) and the Central National Council for Applied Scientific Research in the Netherlands (T.N.O.), by the Municipality of Amsterdam and by several industries.

Summary

In the first part of this report it is shown that generating and analysing can be done elegantly with the aid of ALGOL 60. In the second part a method is described for translation between languages of which we can construct sufficiently similar grammars, by means of a one way simultaneous grammar. In this course a form of PSG will be introduced, essentially equivalent to a CF PSG, but better suited for natural languages. We will demonstrate this method with two ALGOL programs, a small translator English --> Dutch and a somewhat larger translator English --> German.

0.	Introduction	1
1.0	Generative grammar	2
1.1	Program I	4
1.2	Output I	5
2.0	Analytic grammar	6
2.1	Example	9
2.2	Output II	11
3.0	Translating	12
3.1	Program II and input I	15
3.2	Output II	20
4.0	Affixes	24
4.1	Affix PSG	27
4.2	Affix PSG's and CS PSG's	29
4.3	Ambiguity	31
5.0	Translator English German	33
5.1	Program III	39
5.2	Some test results	51
5.3	Input II and output IV	52
6.	References	55

0. Introduction

A context free Phrase Structure Grammar (PSG) consists of a set of terminal symbols or words Vt, a set Vn of nonterminal symbols or grammatical categories, a special nonterminal symbol S called the initial symbol and a set F of rewriting rules of the form:

$$\phi \dashrightarrow \psi$$

$$\text{or } \phi \dashrightarrow \chi\lambda$$

where ϕ is a nonterminal symbol, ψ, χ and λ either terminal or nonterminal symbols. " $\chi\lambda$ " is the concatenation (juxtaposition, Verkettung) of χ and λ. The sign "-->" can be read as "is defined as", "consists of" or "becomes". There can be more than one rule with the same left part. There is at least one rule with as left part the initial symbol. We will write terminal symbols with CAPITAL LETTERS and nonterminal symbols with small letters.

A more manageable equivalent form for a Context Free (CF) PSG is the Backus Normal Form (BNF). All rules with the same left part are gathered together, taking as right part the right parts of the original rules, separated by "/" which should be read as "or". We will call those alternatives. The number of grammatical categories is drastically reduced by allowing more involved right parts as alternatives, concatenations of one or more terminal or nonterminal symbols. From now on a PSG is understood to be in BNF.

We will introduce the symbol "==>" by the following definition: If a grammar contains a rule α --> β, then for any, possibly empty, strings ϕ and ψ of nonterminal or terminal symbols $\phi\alpha\psi \Longrightarrow \phi\beta\psi$. Furthermore for any ϕ, χ and ψ: if $\phi \Longrightarrow \chi$ and $\chi \Longrightarrow \psi$ then $\phi \Longrightarrow \psi$.

" $\phi \Longrightarrow \psi$ " can be read: " ψ is derivable from ϕ". Any string consisting of terminal symbols only will be called a sentence. The language L(G) of a certain grammar G(Vt, Vn, S, F) is the set, possibly infinite, of all sen-

tences that are derivable from the initial symbol S with the aid of G.
The sentences belonging to L will be called grammatical.
The process of constructing a grammatical sentence from S by applying rewriting rules is called generating a sentence.
By analyzing a sentence is understood finding out whether the sentence is grammatical and, if so, giving a derivation (analysis) of it.

It can be shown that it is always possible to check whether a given sentence is grammatical with respect to a certain CF PSG.

1. Generative grammar

Let us consider the following simple grammar, with "sent" as an initial symbol:

sent --> noun verb noun
verb --> LOVES
noun --> JOHN / MARY

We will use it as a generative grammar. We start with "sent". We can do only one thing: "noun verb noun". The first noun gives us the choice between JOHN and MARY. Choosing MARY we have: "MARY verb noun". As a verb we can only take LOVES resulting in: "MARY LOVES noun". Again we experience l'embarras du choix between JOHN and MARY. Both "MARY LOVES MARY" and "MARY LOVES JOHN" are grammatical sentences, though for semantic reasons we might prefer the latter.

Performing this hand simulation has already given us some ideas about the realization of a generative grammar and presented us with the difficulty of choosing between various alternatives. "a sent must consist of noun followed by verb followed by noun". A programmer will interpret this as: "sent consecutively calls noun, verb and noun". Or in ALGOL:

```
procedure sent; begin noun; verb; noun end;
procedure verb; WRITE(≮LOVES≯);
procedure noun; if criterion then WRITE(≮JOHN≯)
                                   else WRITE(≮MARY≯);
```

For typographical reasons the symbols "≮" and "≯" stand for "6" and "9" respectively. " " and line feed occurring within string quotes denote the corresponding typographical symbols.
WRITE(u) is the procedure that writes the string u. If Pjohn is a real number between 0 and 1, RANDOM a real procedure delivering a random number between 0 and 1, then criterion could look like this:

```
boolean procedure criterion; criterion:= RANDOM < Pjohn;
```

We code a slightly larger grammar into program I from which the grammar with probabilities can easily be reconstructed. Perhaps the procedure nounphrase needs elucidating. It is a transcription of:

```
nounphrase --> (.25) adjective nounphrase /
               (1.0) nounpart
```

This makes the probability of having exactly n adjectives in front of a substantive $P(n) = .75 \times 4 \uparrow -n$, which seems a reasonable approximation, the character of the actual distribution being unclear. At the same time this furnishes an example of a recursively used rule, making the length of a sentence potentially infinite.
The program was run on the ELECTROLOGICA X1 of the ERC, Utrecht, using the compiler written by E. W. Dijkstra and J. A. Zonneveld.
It makes use of the following undeclared procedures:

PUTEXT1(u) causes the string u to be punched.

PUNLCR punches a new line carriage return.

SETRANDOM(x), $0 \leq x < 1$, is a preparation for RANDOM, which delivers a real random number, $0 \leq$ RANDOM < 1.

XEEN(2047) delivers the contents of the rightmost eleven switches on the console of the X1.

With +181 in the switches the program gave output I.

```
begin   comment jungle generator    1.1 Program I;
        real rr;
        real procedure R; R:= rr:= RANDOM;
        procedure P(u); string u; PUTEXT1(u);

        procedure sentence;
        begin    subject; predicate end;
        procedure subject;
        if R < .8 then subst else subname;
        procedure subst;
        begin    article; noun phrase end;
        procedure noun phrase;
        if R < .25 then begin adje; noun phrase end else nounpart;
        procedure nounpart;
        if R < .25 then begin noun; relsentence end else noun;
        procedure predicate;
        if R < .2 then begin modifier; modpredicate end else modpredicate;
        procedure modpredicate;
        begin    verb; object end;
        procedure relsentence;
        begin    P(⊰, that⊱); predicate end;
        procedure object;
        if R < .8 then subst else objname;
        procedure adje;
        if R < .2 then begin adverb; adjective end else adjective;
        procedure subname;
        if R < .25 then P(⊰ he⊱) else if rr < .5 then P(⊰ Jim⊱)
        else if rr < .75 then P(⊰ Mary⊱) else P(⊰ she⊱);
        procedure objname;
        if R < .25 then P(⊰ him⊱) else if rr < .5 then P(⊰ Jim⊱)
        else if rr < .75 then P(⊰ Mary⊱) else P(⊰ her⊱);
        procedure modifier;
        if R < .33 then P(⊰ always⊱) else if rr < .67 then P(⊰ often⊱)
        else P(⊰ never⊱);
        procedure article;
        if R < .45 then P(⊰ a⊱) else P(⊰ the⊱);
        procedure noun;
        if R < .26 then P(⊰ boy⊱) else if rr < .48 then P(⊰ tree⊱)
        else if rr < .74 then P(⊰ girl⊱) else P(⊰ bear⊱);
        procedure adjective;
        if R < .25 then P(⊰ little⊱) else if rr < .5 then P(⊰ meek⊱)
        else if rr < .75 then P(⊰ big⊱) else P(⊰ bad⊱);
        procedure adverb;
        if R < .5 then P(⊰ very⊱) else P(⊰ rather⊱);
        procedure verb;
        if R < .25 then P(⊰ sees⊱) else if rr < .5 then P(⊰ likes⊱)
        else if rr < .75 then P(⊰ dreams about⊱) else P(⊰ eats⊱);

        SETRANDOM(XEEN(2047) / 2048);
NZ:     PUNLCR; sentence; PUNLCR;
        if XEEN(-0) > 0 then goto NZ
end
```

1.2 Output I

the bear, that sees the bear likes a bad bear

the bear likes him

the boy likes the bear

the girl dreams about a bear

a girl never sees the girl

Jim sees Mary

Mary sees a bear

a boy, that sees the very big tree always likes a boy, that always eats a tree

Jim likes the boy

the girl, that sees a bear sees a boy, that sees the tree

the tree always sees a very meek bear, that dreams about the girl, that sees the tree, that eats Mary

a tree sees the bear, that eats the girl

he never sees a girl

the bear sees the girl

the boy likes him

a boy sees a tree

a girl likes the boy

a boy, that always likes him likes Jim

she dreams about the boy, that sees Jim

she likes the boy

a bear never dreams about the girl, that sees him

the bad tree dreams about Jim

the bear, that likes Mary sees the girl, that likes the big girl, that eats the girl

2.0 Analytic grammar

The first method we can think of for analysing sentences is the application of the rules in the reverse direction. The inverse of the rule " $\alpha \rightarrow \beta \gamma$ " will then be:

> If the i'th word of a sentence is a β and the i + 1'th a γ, then put the i'th word equal to α and erase the i + 1'th.

This turns out to be a very cumbersome method. The following method appears to be simpler and more elegant:

> Try to generate a sentence that matches the given sentence word by word. (analysis by synthesis, tentative generation).

Our first thoughts go out to the following as a realization of the analyser for the JohnLovesMary grammar:

```
boolean procedure sentence; sentence:= noun ∧ verb ∧ noun;
boolean procedure noun; noun:= JOHN ∨ MARY;
boolean procedure verb; verb:= LOVES;
```

LOVES assumes the value true if and only if the next word of the sentence equals "love".

This approach works for this very simple grammar, but soon we get into problems, as for instance with:

```
nounphrase - > adjective nounphrase / nounpart
```

with transcription:

```
boolean procedure nounphrase;
nounphrase:= (adjective ∧ nounphrase) ∨ nounpart;
```

Nounphrase will call itself until eternity. Another problem concerns what is meant by "the following word of the sentence". A better approach: we declare a global boolean b and integer p and write:

```
procedure sentence;
begin   PUTEXT1(‹sentence›); p:= 1;
        noun; if b then verb; if b then noun
end;
```

```
procedure verb;
begin   PUTEXT1(⊰verb⊱); match(⊰loves⊱) end;
procedure noun;
begin   PUTEXT1(⊰noun⊱); match(⊰john⊱);
        if ¬b then match(⊰mary⊱)
end;
```

The procedure match(u) does the following: p points to the next word of the sentence. If this word is equal to the string u then b is made true and p incremented by 1, else b is set to false. Finally u is punched out, followed by "match" or "fail" according to whether b is true or false. Now nounphrase is realizable as follows:

```
procedure nounphrase;
begin   PUTEXT1(⊰nounphrase⊱); adjective;
        if b then nounphrase else nounpart
end;
```

A non-recursive version is possible too, e. g.:

```
procedure nounphrase;
begin   PUTEXT1(⊰nounphrase⊱);
LABEL: adjective; if b then goto LABEL; nounpart
end;
```

We can be presented with the following problem:

```
sentence --> subject predicate
subject --> HE quality / HE
quality --> adverb READY TO GO
predicate --> adverb STOPS
adverb --> ALWAYS / NEVER
```

Our mechanism would be able to analyze

HE ALWAYS READY TO GO NEVER STOPS

but not

HE ALWAYS STOPS

for in the second sentence, when it has failed to find a quality, it will find "stops" as a predicate instead of "always stops". To cope wih this "common constituent problem" every procedure that returns with b = false will have to restore p to its value at the moment of call. Therefore we make use of the concept of locality, thus:

```
procedure quality;
begin   integer n; n:= p; PUTEXT1(⊰quality⊱);
        adverb; if b then match(⊰ready to go⊱);
        if ¬b then p:= n
end;
```

For a rule like

subject --> HE quality / HE

We will use the shorter notation

subject --> HE (quality)

quality put between brackets to indicate that it is optional. It can be programmed:

```
procedure subject;
begin   integer n; n:= p; PUTEXT1(⊰subject⊱);
        match(⊰he⊱); if ¬b then goto F;
        quality; b:= true; goto E;
F:      p:= n; E:
end;
```

In the same way we write

modverb --> (modifier) verb

for modverb --> modifier verb / verb

2.1 Example

The following grammar II, which makes use of the shorter notation, has been programmed in ALGOL as an analyser.

sentence --> subject predicate
subject --> subst / subname
subst --> article nounphrase
nounphrase --> adje nounphrase / nounpart
nounpart --> noun (relsentence)
predicate --> modverb object
modverb --> (adverb) verb
relsentence --> ,WHO predicate / ,THAT predicate
object --> subst / objname
adje --> (modifier) adjective
subname --> BADENPOWELL / JOHNWAYNE /
DAVIDLIVINGSTONE / HE
objname --> BADENPOWELL / JOHNWAYNE /
DAVIDLIVINGSTONE / HIM
adverb --> ALWAYS / OFTEN / NEVER
article --> A / THE
noun --> RHINOCEROS / GORILLA / CANNIBAL / MISSIONARY
adjective --> FAT / SLEEPY / NOISY / ABOMINABLE
modifier --> NONETOO / RATHER
verb --> SEES / SMELLS / KILLS / EATS

As the program is largely incorporated in the translator program English --> Dutch it is not reproduced here. The program was run on the X1 of the ERC in UTRECHT. Analyzing "JOHN WAYNE SMELLS THE SLEEPY MISSIONARY" it produced output II.

One can see clearly, not only what the structure is, but what it is not. If the sentence does not contain a relative sentence we prefer to see no trace of the attempt to find one. We will realize this by means of an output stack with pointer w.
Every procedure should start with Q(name) instead of PUTEXT1(name), where Q does not punch directly, but stores the name under control of the pointer w.
At the end of the procedure, w resumes its value at the moment of call if b turns out to be <u>false</u>, thus erasing all traces of irrelevant attempts. Furthermore we declare a global integer as, that counts the "number of spaces". This number is incremented by each call of Q and decremented at the end of each procedure by the

```
procedure UQ; as:= as - 1;
```

With this apparatus the analysis of JOHN LOVES MARY looks like this:

```
sentence
 noun
  JOHN
 verb
  LOVES
 noun
  MARY
```

The complete text of quality is now

```
procedure quality;
begin   integer n, m; n:= p; m:= w; Q(⟨quality⟩);
        adverb; if b then match(⟨ready to go⟩);
        if ¬b then begin p:= n; w:= m end; UQ
end;
```

For the detailed realization of the string handling procedures, see the translator program (3.1).

Analyzing according to a PSG proves to be quite feasible in ALGOL: the lack of string handling facilities is partly made up for by the elegance of its procedure concept.

2.2 Output II

johnwayne smells the sleepy missionary.

```
sentence
subject
subst
article
a           fail
the           fail
subname
badenpowell           fail
johnwayne           match
predicate
modverb
adverb
always           fail
often           fail
never           fail
verb
sees           fail
smells           match
object
subst
article
a           fail
the           match
nounphrase
adje
modifier
nonetoo           fail
rather           fail
adjective
fat           fail
sleepy           match
nounphrase
adje
modifier
nonetoo           fail
rather           fail
adjective
fat           fail
sleepy           fail
noisy           fail
abominable           fail
nounpart
noun
rhinoceros           fail
gorilla           fail
cannibal           fail
missionary           match
relsentence
,that           fail
```

correcte zin.

3.0 Translating

The simplest way of translation is word-by-word translation. The grammar contains a list of words in the first language with one or more equivalents in the second. For the English word form "go" we should have to list as Dutch equivalents:

go --> ga, gaat, gaan; go (japans damspel).

We can discriminate between those alternatives taking into account the grammatical clues we can find in the sentence. Mechanizing such a process gives a more ambitious translator, for which we need an analysis of the sentence that is to be translated.

In translating, the English word order would be preserved. But consider:

JOHN NEVER STOPS

and its Dutch translation

JOHN STOPT NOOIT

The order in Dutch is clearly different from that in English. The English grammar has

verb --> (modifier) verbform

where the Dutch grammar has

verb --> verbform (modifier)

We can solve this order-problem by analyzing in the normal English order, but changing the order during output. We need a notation and a mechanism for that.

Everybody knows what I mean [this | by]. In a linear notation we will denote it < this | by >. A PSG of the first language, thus annotated with inversion brackets and containing for every word in the first language its equivalent in the second language will be called a <u>simultaneous grammar</u> from the first language into the second.

Now the question is raised: can any permutation of n elements be reached with (possibly nested) inversion brackets? The answer is no, as is shown by the counter-example (n = 4): 3 1 4 2. Exhaustively trying all combinations of brackets, one will find that it can not be permuted to 1 2 3 4.

We could try to define the operation of those brackets in such a way that not properly nested use becomes meaningful, thus:

reading a string from left to right, the first inversion opening bracket corresponds to the first inversion middle bracket after that, and the first inversion end bracket after that middle bracket. In reading the string, corresponding brackets are removed, performing the inversions one after the other. (An alternative definition could be given with "left" and "right" interchanged.)

Now we can permute:

$\ll 3 \mid 1 > 4 \mid 2 >$ to $1 < 3 \;\; 4 \mid 2 >$ to $1 \;\; 2 \;\; 3 \;\; 4$.

Again the question is raised: can any permutation of n elements be reached with inversion brackets as defined? The reader may tax his ingenuity by either disproving it or finding an algorithm which does position the brackets. The author has spent a whole night trying in vain to find a solution for

6 1 4 2 7 5 3 to 1 2 3 4 5 6 7 .

For the time being we will allow only properly nested inversions.

In a program, the inversion opening bracket will be represented by a call of the procedure invopen, a middle bracket by invmiddle and an end brac-

ket by invend. Using these the transcription of

```
verb --> < (modifier) | verbform >
```

will be

```
procedure verb;
begin   integer m, n; n:= p; m:= w; Q(≮verb≯);
        invopen; modifier; invmiddle; verbform; invend;
        if ¬b then begin p:= n; w:= m end; UQ
end;
```

When there is no modifier the output is the structure "< | verbform >", which is equivalent to "verbform". Invopen, invmiddle and invend put into the output stack a note for the output program, that will effect the inversion. Their use may be nested, as is shown by one of the translated sentences.

The translation proceeds as follows: for each English word form in a certain grammatical class, the equivalent Dutch word form is listed. The procedure match gives besides the English word its appropriate Dutch translation. Thus, the Dutch sentence is generated parallel to the analysis of the English sentence.

Input I consists of those two vocabularies, followed by the list of names of grammatical classes, followed by some sentences. The program, program II, is a good demonstration of the techniques we have expounded thus far, and a good introduction to a moderately complex translator: The small English --> German translator (5.0).

```
begin   comment jungle translator, K. Koster, 110864      Program II;
        integer r, n, l1, l2, l3, l4, p, w, as, i, j, k; boolean b, op;
        integer array A[0 : 30], W[0 : 300], T[0 : 1000], O[0 : 500];

        integer procedure R;
        begin l: r:= R:= RE7BIT;
                 if r = 26 ∨ r = 122 ∨ r = 124 ∨ r = 127
                    ∨ (r = 16 ∧i = 0) ∨ r = 0 then goto l
        end R;

        boolean procedure o; o:= XEEN(2) = 2;

        procedure P(i); integer i;
        begin    switch PS:= p1, p2, p4, p7, p8, pe, pe, pe, psp, p3, p5, p6, p9,
                              pnl, pe, pe, p0, pt, pv, pw, pz, pe, pe, pe,
                              pe, ps, pu, px, py, pe, pe, ptb, pmin, pl, pn,
                              po, pr, pe, pe, pe, pj, pk, pm, pp, pq, pcom,
                              pe, pe, pa, pb, pd, pg, ph, ppnt, pe, pe, pplus,
                              pc, pee, pf, pi, pe, pe, pe;
                 procedure P(u); string u;
                 begin    if XEEN(8) = 8 then PUTEXT1(u) else PRINTTEXT(u);
                          goto pe
                 end;
                 goto if i = 0 then pe else PS[i : 2 + 1];
                 p1: P(≮1≯); p2: P(≮2≯); p3: P(≮3≯); p4: P(≮4≯); p5: P(≮5≯);
                 p6: P(≮6≯); p7: P(≮7≯); p8: P(≮8≯); p9: P(≮9≯);
                 p0: P(≮0≯); pa: P(≮a≯); pb: P(≮b≯); pc: P(≮c≯); pd: P(≮d≯);
                 pee: P(≮e≯); pf: P(≮f≯); pg: P(≮g≯); ph: P(≮h≯);
                 pi: P(≮i≯); pj: P(≮j≯); pk: P(≮k≯); pl: P(≮l≯); pm: P(≮m≯);
                 pn: P(≮n≯); po: P(≮o≯); pp: P(≮p≯); pq: P(≮q≯);
                 pr: P(≮r≯); ps: P(≮s≯); pt: P(≮t≯); pu: P(≮u≯); pv: P(≮v≯);
                 pw: P(≮w≯); px: P(≮x≯); py: P(≮y≯); pz: P(≮z≯);
                 psp: P(≮ ≯); ptb: P(≮      ≯); pmin: P(≮-≯); pcom: P(≮,≯);
                 pplus: P(≮+≯); ppnt: P(≮.≯); pnl: P(≮
≯); pe:
        end;

        procedure OUTPUT;
        begin    integer c, d, e;
                 for c:= 0 step 1 until w - 1 do
                 begin    d:= O[c]; if d < 0 then
                          begin    if d < -30767 then c:= d + 32767 else
                                   if d < -256 then print(d) else
                                   begin    if d < -127 then
                                            begin    PUNLCR; d:= d + 128 end;
                                            for e:= 1 step 1 until -d do P(16)
                                   end
                          end               else
                          begin    e:= d : 128; P(d - 128 × e);
                                   d:= e : 128; P(e - 128 × d); P(d)
                          end
                 end
        end;
```

```
procedure PONS(i, j); integer i, j;
begin    integer a; O[w]:= -j; w:= w + 1;
         if i ≥ 0 then for a:= W[i] step 1 until W[i + 1] - 1 do
         begin    O[w]:= T[a]; w:= w + 1 end
end;

procedure INPUT;
begin    W[0]:= 0;
LI:      lees; if r ≠ 107 then goto LI
end;

procedure lees;
begin    i:= 0;
lees1:   if R = 16 ∨ r = 107 then goto codeer;
         A[i]:= r; i:= i + 1; goto lees1;
codeer:  for j:= 0, j + 3 while j < i do
         begin T[k]:= A[j]   (if j+ 1 < i then A[j + 1] × 128 else 16256)
                  + if j + 2 < i then A[j + 2] × 16384 else 2080768;
                  k:= k + 1
         end;
         n:= n + 1; W[n]:= k; if g then PONS(n - 1, 128)
end;

boolean procedure g; g:= XEEN(1) = 1;

procedure match(i); integer i;
begin    if p < n then equal(p, i) else
         if p > n then b:= false else
         if op then b:= false else
         begin lees; op:= r = 107; equal(p, i) end;
         if b then
         begin    p:= p + 1; PONS(i, 128 + as); PONS(i + 13, 10) end
              else if g then PONS(i, 126 + as)
end;

procedure equal(i, p); integer i, p;
begin    integer j, k, l, m, t;
         j:= W[i]; k:= W[i + 1] - 1;
         l:= W[p]; m:= W[p + 1] - 1;
         if k - j ≠ m - l then begin b:= false; goto endeq end;
         for t:= k - j step -1 until 0 do
         begin    if T[j + t] ≠ T[l + t] then
                  begin    b:= false; goto endeq end
         end;
         b:= true; endeq:
end;

procedure Q(i, u); integer i; string u;
begin    if o then PONS(14 + i, 128 + as)
              else if g then begin SPACE(1); PRINTTEXT(u) end;
         as:= as + 1
end;

procedure UQ; as:= as - 1;
```

```
procedure invopen; PONS(-1, 258);

procedure invmiddle; PONS(-1, 257);

procedure invend;
begin    integer i, j; i:= w - 1;
AA:      if O[i] = -257 then O[i]:= w - 32767
                            else begin i:= i - 1; goto AA end;
         j:= i - 1;
BA:      if O[j] = -258 then O[j]:= i - 32767
                            else begin j:= j - 1; goto BA end;
         O[w]:= j -32767; w:= w + 1
end;


procedure sentence;
begin    Q(0, <se>);
         subject; if b then predicate; UQ
end;
procedure subject;
begin    Q(1, <sj>); subst;
         if ¬b then subname; UQ
end;
procedure subst;
begin    integer m, n; m:= w; n:= p; Q(2, <su>);
         article;
         if b then noun phrase;
         if ¬b then begin p:= n; w:= m end; UQ
end;
procedure noun phrase;
begin    integer m, n; m:= w; n:= p; Q(3, <nf>);
ADLST:   adje; if b then goto ADLST;
         nounpart;
         if ¬b then begin p:= n; w:= m end; UQ
end;
procedure nounpart;
begin    integer n, m; m:= w; n:= p; Q(4, <np>);
         noun;
         if b then begin relsentence; b:= true end
              else begin p:= n; w:= m end; UQ
end;
procedure predicate;
begin    integer m, n; m:= w; n:= p; Q(5, <pr>);
         modverb; if b then object;
         if ¬b then begin  p:= n; w:= m end; UQ
end;
procedure modverb;
begin    integer m, n; m:= w; n:= p; Q(6, <mv>);
         invopen; adverb; invmiddle; verb; invend;
         if ¬b then begin p:= n; w:= m end; UQ
end;
```

```
procedure relsentence;
begin    integer m, n; m:= w; n:= p; Q(7, ≮rs≯);
         match(0); if ¬b then match(24);
         if b then
         begin    adverb; invopen; verb;
                  if b then
                  begin    invmiddle; object; invend end
         end; if ¬b then begin p:= n; w:= m end; UQ
end;
procedure object;
begin    Q(8, ≮ob≯); subst;
         if ¬b then objname; UQ
end;
procedure adje;
begin    integer m, n; m:= w; n:= p; Q(9, ≮aj≯);
         modifier; adjective;
         if ¬b then begin p:= n; w:= m end; UQ
end;
procedure subname;
begin    integer m; m:= w; Q(10, ≮sn≯);
         match(1); if ¬b then match(2); if ¬b then match(3);
         if ¬b then match(4);
         if ¬b then w:= m; UQ
end;
procedure objname;
begin    integer m; m:= w; Q(11, ≮on≯);
         match(1); if ¬b then match(2); if ¬b then match(3);
         if ¬b then match(25);
         if ¬b then w:= m; UQ
end;
procedure adverb;
begin    integer m; m:= w; Q(12, ≮av≯);
         match(5); if ¬b then match(6); if ¬b then match(7);
         if ¬b then w:= m; UQ
end;
procedure article;
begin    integer m; m:= w; Q(13, ≮ar≯);
         match(8); if ¬b then match(9);
         if ¬b then w:= m; UQ
end;
procedure noun;
begin    integer m; m:= w; Q(14, ≮no≯);
         match(10); if ¬b then match(11); if ¬b then match(12);
         if ¬b then match(13);
         if ¬b then w:= m; UQ
end;
procedure adjective;
begin    integer m; m:= w; Q(15, ≮ad≯);
         match(14); if ¬b then match(15); if ¬b then match(16);
         if ¬b then match(17);
         if ¬b then w:= m; UQ
end;
```

```
        procedure modifier;
        begin   integer m; m:= w; Q(16, ⊀mf⊁);
                match(22); if ¬b then match(23);
                if ¬b then w:= m; UQ
        end;
        procedure verb;
        begin   integer m; m:= w; Q(17, ⊀vb⊁);
                match(18); if ¬b then match(19); if ¬b then match(20);
                if ¬b then match(21);
                if ¬b then w:= m; UQ
        end;

INGRAM: n:= k:= w:= 0; INPUT; 13:= n; INPUT; 14:= n;
        INPUT; 11:= n; 12:= k; b:= true;
        if g then begin NLCR; for i:= 13, 14, 11, 12 do print(i); NLCR end;
BASIS:          PUNLCR; if ¬b then w:= w + 50; comment voor testdoeleinden;
        OUTPUT; PUNLCR;
        if XEEN(4) = 4 then stop; p:= n:= 11; k:= 12; w:= as:= 0;
        PUNLCR; op:= b:= false; sentence; b:= b ∧ op; PONS(-1, -107);
        PUNLCR; PUNLCR; if b then PUTEXT1(⊀correcte zin.⊁)
                              else PUTEXT1(⊀geen welgevormde zin.⊁);
        PUNLCR; goto BASIS
end
```

'INPUT I'

,that badenpowell johnwayne davidlivingstone he always often never a the
rhinoceros gorilla cannibal missionary fat sleepy noisy
abominable sees smells kills eats nonetoo rather ,who him.

,die badenpowell johnwayne davidlivingstone hij altijd vaak nooit een de
neushoorn gorilla kannibaal missionaris vette slaperige lawaaierige
verschrikkelijke ziet ruikt doodt verorbert nietalte nogal ,die hem.

sentence subject subst nounphrase nounpart predicate modverb relsentence
object adje subname objname adverb article noun adjective modifier
verb empty.

the gorilla ,that often kills badenpowell never eats a missionary.

johnwayne smells the fat rhinoceros ,that always eats a gorilla ,
that sees the noisy cannibal.

the nonetoo fat cannibal sees a rather fat missionary.

davidlivingstone never sees a sleepy gorilla ,that kills the
rather abominable noisy missionary.

3.2 Output III

correcte zin.

```
sentence
 subject
  subst
   article
    the             de
   nounphrase
    nounpart
     noun
      gorilla            gorilla
     relsentence
      ,that            ,die
      adverb
       often             vaak
      object
       objname
        badenpowell               badenpowell
      verb
       kills            doodt
 predicate
  modverb
   verb
    eats            verorbert
   adverb
    never             nooit
  object
   subst
    article
     a             een
    nounphrase
     nounpart
      noun
       missionary             missionaris.
```

correcte zin.

```
sentence
 subject
  subname
   johnwayne          johnwayne
 predicate
  modverb
   verb
    smells           ruikt
  object
   subst
    article
     the            de
    nounphrase
     adje
      adjective
       fat           vette
     nounpart
      noun
       rhinoceros           neushoorn
      relsentence
       ,that          ,die
       adverb
        always          altijd
       object
        subst
         article
          a           een
         nounphrase
          nounpart
           noun
            gorilla          gorilla
           relsentence
            ,that           ,die
            object
             subst
              article
               the           de
              nounphrase
               adje
                adjective
                 noisy          lawaaierige
               nounpart
                noun
                 cannibal         kannibaal
            verb
             sees          ziet
       verb
        eats          verorbert.
```

correcte zin.

```
sentence
 subject
  subst
   article
    the           de
   nounphrase
    adje
     modifier
      nonetoo           nietalte
     adjective
      fat           vette
    nounpart
     noun
      cannibal            kannibaal
 predicate
  modverb
   verb
    sees           ziet
  object
   subst
    article
     a           een
    nounphrase
     adje
      modifier
       rather           nogal
      adjective
       fat           vette
     nounpart
      noun
       missionary           missionaris.
```

correcte zin.

```
sentence
 subject
  subname
   davidlivingstone            davidlivingstone
 predicate
  modverb
   verb
    sees           ziet
   adverb
    never           nooit
  object
   subst
    article
     a         een
    nounphrase
     adje
      adjective
       sleepy          slaperige
     nounpart
      noun
       gorilla          gorilla
      relsentence
       ,that          ,die
       object
        subst
         article
          the          de
         nounphrase
          adje
           modifier
            rather          nogal
           adjective
            abominable          verschrikkelijke
          adje
           adjective
            noisy          lawaaierige
          nounpart
           noun
            missionary          missionaris
       verb
        kills          doodt.
```

4.0 Affixes

Consider the following grammar to generate

1) THE GORILLA EATS FRESH PEANUTS
sentence --> subject verb object
subject --> article substantive
object --> adjective substantive
substantive --> GORILLA / PEANUTS
article - > THE
verb --> EATS
adjective --> FRESH

This grammar also covers the sentences

2) THE PEANUTS EATS FRESH PEANUTS
3) THE GORILLA EATS FRESH GORILLA
4) THE PEANUTS EATS FRESH GORILLA

Even if the sentence 3) evokes a rather improbable image, we do not feel any grammatical objection against it; but we do feel one against the sentences 2) and 4). We feel that the verb should follow the subject in number. We can solve the problem by discriminating between singular and plural subjects.

sentence --> subjectsingular verbsingular object /
subjectplural verbplural object
subjectsingular --> article substsingular
subjectplural --> article substplural
object --> adjective substsingular / adjective substplural
verbsingular --> EATS
verbplural --> EAT
substsingular --> GORILLA
substplural --> PEANUTS
article --> THE
adjective --> FRESH

This grammar yields the following sentences:

1) THE GORILLA EATS FRESH PEANUTS
2) THE GORILLA EATS FRESH GORILLA
3) THE PEANUTS EAT FRESH GORILLA
4) THE PEANUTS EAT FRESH PEANUTS

In this way a grammar for a natural language becomes frightfully large and unwieldy, as we have to differentiate each grammatical category according to person, number, gender, time, case, whether it is active or passive, a question or an order and probably a lot of less obvious subdivisions.
One way to keep the grammar small is to have only simple sentences in the grammar and to construct more involved sentences out of them by the application of transformations. But in the case of analysis instead of generation this seems to be putting the cart before the horse: an analysis of the sentence is needed anyway, and the most practical way of defining a transformation is probably a simultaneous grammar from the natural language into itself.

The method we will use to keep our grammar concise and clear is the use of affixes. "verb with the affixes a and b" will be written "verb + a + b". Affixes may be seen as formals in ALGOL procedures (left of "-->" the heading and specifications, right of "-->" the procedure body) or as endings to the names of the grammatical categories, that can be copied and transferred.
A rule "verb + n + p --> auxiliaryverb + n + p infinitive" transforms "verb + singular + third" into "auxiliaryverb + singular + third infinitive". The grammar should contain a rule with a left part of the form:

```
        verb + singular + first -->
                        + second -->
                          third -->
             + plural + first -->
                      + second -->
                      + third -->
```

(The deletion of repeated elements is just an economy in writing down the grammar).

Now we can give the grammar a neater, though completely equivalent form.

```
        sentence --> basicsentence + number
        number - > singular / plural
        basicsentence + n --> subject + n verb + n object
        subject + n --> article substantive + n
        object - > adjective substantive + number
        verb + singular --> EATS
             + plural --> EAT
        article --> THE
        substantive + singular --> GORILLA
                    + plural --> PEANUTS
        adjective - > FRESH
```

As each rule can have only a finite number of affixes, and each affix is a member of a finite set, we can write out an affix PSG into an equivalent CF PSG (by subdividing categories according to the different affixes they bear, for instance adding them as endings to the head word). In fact this is mechanical work that can be done by a computer.
This is an obvious proof for the following:

theorem for every affix PSG there is a strictly equivalent CF PSG.

This means that an affix PSG is not a stronger tool than a normal PSG, but the reduction in size of a grammar by making use of some affixes can be tremendous.

If we allow affixes to be grammatical categories then the proof of the theorem is still valid, for the number of grammatical classes is finite too. A practical instance of this:

progressive + n + p + v --> tobe + n + p v + partpraes

will transform

progressive + n + p + see

into

tobe + n + p see + partpraes

The affix mechanism can be realized in ALGOL by making use of parameters. Grammatical categories are represented by procedures and it is quite correct to use a formal procedure.

4.1 Affix PSG's

We will give a more rigorous mathematical description of an affix phrase structure grammar G(V, S, F, P, M).

The vocabulary V consists of 5 mutually disjoint vocabularies

Vn of nonterminal symbols
Vt of terminal symbols
Vaf of formal affixes
Van of nonterminal affixes
Vat of terminal affixes

S is the special initial symbol, F, P and M three lists of rules.

We will describe the possible structure of F, P and M rules by a PSG, using as a meta-meta-language the notational conventions of the ALGOL report. (We will not use the symbols <, | and > for inversion brackets). We presuppose categories like <member of Vn>, etc.

<F-rule>::= <member of Vn> --> <rightpart>
<rightpart>::= <alternative> / <rightpart> | <alternative>
<alternative>::= <constituent> <alternative> | <constituent>
<constituent>::= <member of Vn> | <member of Vt>

<P-rule>::= <member of Van> --> <P-list>
<P-list>::= <actual affix> / <P-list> | <actual affix>
<actual affix>::= <member of Vat> | <constituent>

<M-rule>::= <member of Vn> <left affix list> --> <M-rightpart>
<left affix list>::= <left affix> + <left affix list> | <left affix>
<left affix>::= <member of Vaf> | <member of Vat>
<M-rightpart>::= <M-alternative> / <M-rightpart> | <M-alternative>
<M-alternative>::= <M-constituent> <M-alternative> | <M-constituent>
<M-constituent>::= <M-headword> · <right affix list> |
<M-headword> | <member of Vt>
<M-headword>::= <member of Vn> | <member of Vaf>
<right affix list>::= <right affix> + <right affix list> | <right affix>
<right affix>::= <any member of V>

If some member of Vaf occurs as a right affix in a rule, it must occur once and only once in the left affix list of the head word as well. The reverse is not necessarily true.
P-rewriting rules have precedence over other rewriting rules, so that a member of Van is immediately rewritten.

M-rewriting rules are applied as follows:

a rule "<member of Vn> + <leftaffix>1 + ... + <leftaffix>n -->"
is applicable to

"<M-headword> + <rightaffix>1 + ... + <rightaffix>m"

if and only if

<member of Vn> equals <M-headword>

$m = n$

and for all i, $1 \leq i \leq n$,

either <leftaffix>i is a member of Vaf

or both <leftaffix>i and <rightaffix>i are members of Vat

and <right affix>i equals <leftaffix>i.

If the rule is applicable rewriting of the M-constituent takes place, into which is substituted - at every occurence in the right part of the rule of some formal affix corresponding to the j'th left affix of the rule - the j'th right affix of the M-constituent. (cf. actual-formal correspondence in ALGOL).

This definition of affix PSG is best suited to generation purposes. In (5.0) we will introduce M-rules with formal affixes in the right part that are not matched by a left affix.

4.2 Affix PSG's and Context Sensitive PSG's

It must be kept in mind that every M-rule in our grammar in effect represents a number of context free rules (F-rules). As long as we have a limited number of affixes we could write out the affix grammar as a CF PSG, even though an affix grammar has some properties we would intuitively attribute to a Context Sensitive grammar or to a discontinuous grammar. It can handle constructions like "both ... and ..." (matched pairs) and "word-by-word" (repetitions).

But if we allow indices to be taken from an infinite set, as for instance the natural numbers, we should get an infinite number of rules. In order to do this we should have at least one P-rule with an infinitely long P-list. We can keep the number of rules finite by allowing calculation, in the form of affix expressions, but this can give us something stronger than a CF PSG.

Take for example the non context free (context sensitive with erasing) language consisting of sentences of the form

A[n] B[n] A[n] (n a's followed by exactly n b's, followed by exactly n a's, n = 1, 2, ...,n, ..).

A context sensitive grammar for this language is

initial symbol S
S --> A B r A
B r A --> B A / q B B A A
B q B - > q B B
A q B --> A A B r
B r B --> B B r

Like busy bees the constituents q and r travel across the half-formed sentence, adding A's and B A's till the process breaks off. As a model of the way that our brain recognizes or generates sentences of this kind it is clearly preposterous.

Now consider the following affix grammar involving calculations. (Affix expressions, put between square brackets, are considered members of Van).

M: sentence --> basicsentence + positiveinteger
P: positiveinteger --> 1, 2, 3, ... , k, ..
M: basicsentence + n --> rowofas + n rowofbs + n rowofas + n
rowofas + 1 --> A
+ n --> A rowofas + [n - 1]
rowofbs + 1 --> B
+ n - > B rowofbs + [n - 1]

A study of this most general kind of affix grammars seems worth while. That affix grammars are a perfectly natural tool for treating natural languages will become obvious. MEERTENS has done some (unpublished) work on the use of affix grammars in musical composition. His results seem to indicate the necessity of allowing affix expressions in those parts of the grammar connected with rhythm and melody.

4.3 Ambiguity

This method of analysis is sensitive to various kinds of grammatical ambiguity. One kind of ambiguity, the common constituent problem, is solved by making use of the pointer p (2.0). A more serious kind of ambiguity, due to a misordering of the alternatives, is the following:

subst --> noun / noun relsentence

This rule will never detect a relsentence. It should be reordered to:

subst --> noun relsentence / noun

putting the more complicated alternative first. Or, using a more practical notation:

subst --> noun (relsentence)

This shows that some care must be taken in programming a PSG as an analyser. Take for instance 2.4.1 of the ALGOL REPORT:

identifier --> letter / identifier letter / identifier digit

The rule, if programmed this way, will incorrectly analyse "q1", for it is ended as soon as the q has been found. If we reorder it we will find that the program never stops, due to the left recursivity of the rule. We have to rewrite it in a right-recursive form:

identifier --> letter (identifiertail)

identifiertail --> letter (identifiertail) / digit (identifiertail)

The form in the report was meant for definition, not analysis purposes.

Even if we construct our grammar with a wary eye on ordering and right recursivity there may be ambiguities with which this simple approach can not cope.

object --> subst (nextobject)
nextobject --> AND object
sentence --> basicsentence (nextsentence)
nextsentence --> AND s entence
basicsentence --> subject SAW object / subject WERE IN BLOOM
subject --> I / subst
subst --> THE GARDENS / THE ROSES

Analysing the sentence

I SAW THE GARDENS AND THE ROSES WERE IN BLOOM

we will not find two sentences separated by AND but a sentence

I SAW THE GARDENS AND THE ROSES

followed by some ungrammatical nonsense. Even though the grammar is quite capable of generating the given sentence, it can not analyse it. This sentence displays another ambiguity: to the English speaking reader it can be equivalent to

I SAW THAT THE GARDENS AND THE ROSES WERE IN BLOOM

An ideal analyzer would in this case give more than one, all possible analyses. The method of analysis by synthesis as proposed here will give at most one analysis - in the case of multiple possibilities the one that comes first according to the implicit ordering of the grammar.

In the definition of affix PSG a right part consists of a number of alternatives, each consisting of a concatenation. We can wonder exactly where grammatical ambiguity originates. It is clear that in concatenations no ambiguity can arise: either all the constituents are present in that order or analysis fails. But, by choosing between a number of alternatives it is possible to

decide prematurely upon some alternative. This problem is partly solved by trying the more involved alternatives before the simpler. But that is exactly the reason why our last example went wrong.

The ideal analyzer should exhaustively try all alternatives, even after one has been found that would suffice. This can be done by duplicating the stack of the ALGOL executive program, or preserving it in some less drastic way. For reasons of speed and simplicity the programs in this report will not attempt this, even if the present system is not fool proof. Still, many problems can be overcome by investing some thought upon the ordering of the grammar. So for the moment we will not be concerned with grammatical ambiguity, not to mention non grammatically decidable semantical ambiguity.

5.0 Translator English --> German

We will now describe a translator program in ALGOL that translates sentences from English into German. The translator is equipped with a rather rudimentary verb mechanism, but a rather elegant noun mechanism. It is a small translator and the author is busy expanding it to a more practical size. Still, it gives within a small scope an idea of the practicality of affixes and inversions.

The reason for choosing those two languages: German has lots of endings, English has few, and there are some interesting word order problems too. The two languages are just similar enough to make translation possible and just dissimilar enough to make translation interesting.

During analysis, values have to be substituted for the different affixes in the rules. The most direct way would be to try all combinations. The program for the following rule would be:

```
class --> part + affix1 + affix2
affix1 --> a1 / a2 / ..... / an
affix2 --> b1 / b2 / .... / bm
  procedure class
  begin    integer affix1, affix2;
           for affix1:= a1, a2, .. , an do
           for affix2:= b1, b2, .. , bm do
           begin part (affix1, affix2); if b then goto E end; E:
 end;
```

For efficiency reasons we will use an other approach. If in the right part of a rule some formal affix occurs, which does not correspond to any left affix of the rule, we assume this affix to be filled in implicitly.
What this means will become clear in the commentary interspersed with the part of the grammar that is to follow.

sentence --> basicsentence (nextsentence)
nextsentence --> connective sentence
basicsentence --> subject + n + p < predicate + n + p /
circumstance < subject + n + p predicate + n + p

The inversion bracket has to effectuate the discrimination in the German word order between "der Mann sieht mich im Garten" and "im Garten sieht der Mann mich". A third word order will be found in the relsentence: "den der Mann im Garten sieht" or "der mich im Garten sieht".
Subject implicitly gives a value to n and p (number and person), so that predicate is entered with n and p already known.

subject + n + p --> < nounphrase + n + p + g + 1 | (TOO) >

Nounphrase should fill in number, person and gender, but the case is prescribed to be one.

object --> < nounphrase + n + p + g + 4 | (TOO) > (nextobject)

n, p and g are to be filled in implicitly.

```
nextobject --> connector object
nounphrase + n + 3 + g + c --> nounpart + n + g + c
                          (relsentence + n + 3 + g) (circumstance)
           + n + p + g + c --> perspron + n + p + g + c
                          (relsentence + n + p + g) (circumstance)
```

Nounpart should give a value to n and g, whereas p is known to be 3. Personal pronoun has to fill in all three.

```
nounpart + n + g + c -->
          THE < noun + n + g + c + ending1 | endingofder + n + g + c > /
          A < noun + n + g + c + ending2 | endingofein + n + g + c > /
          noun + n + g + c + ending3
```

Noun should give a value to n and g. Ending1, 2 and 3 will generate the correct endings of the german adjectives, and are passed on as formal procedures.

```
noun + n + g + c + e -->
          (modifier) adjective < noun + n + g + c + e | e + n + g + c > /
          subst + n + g + c
```

Now subst has to give values to n and g. Simplified, subst looks like this:

```
procedure subst(n, g, c); integer n, g, c;
begin   n:= singular; g:= masculine;
        match(man, mann); if b then goto E; g:= feminine;
        match(woman, frau); if b then goto E; g:= neutral;
        match(child, kind); if b then goto E;
        n:= plural; g:= masculine;
        match(men, maenner); if b then goto E; g:= feminine;
        match(woman, frauen); if b then goto E; g:= neutral;
        match(children, kinder); if ¬b then goto END;
E:      ending of german substantive (n, g, c); END:
end;
```

The procedure match is equipped with two parameters; if a word equal to the first parameter happens to be under the input pointer, the second parameter is put into the output stack, p is incremented and b becomes *true*; otherwise b becomes *false*.
The translation of the subject "a rather small city too" will be:
"< ein < ziemlich gross < Stadt | e > | e > | auch >", or without brackets: "auch eine ziemlich grosse Stadt".
The problem that has given rise to this not so obvious mechanism is the fact that the endings can only be produced when the substantive has been located. This recursive definition seems quite powerful. With a very small addition it could cope also with "a small and beautiful city".

In a practical case, using a grammar that contains thousands of substantives, the procedure subst should not really try to match in turn all substantives listed. Rather some preprocessing program should construct for every word in the input sentence a list of possible grammatical categories, together with the translation and the value of the affixes. The procedure subst only searches the list under the pointer for some translation marked as a substantive, and takes over the values of the affixes. This prevents much double work, especially in the case of multiple analyses. Thus translation time will go up about logarithmically with the number of words in the grammar instead of linearly. In programming the preprocessor, one can benefit from the tremendous effort put into the construction of vocabularies and stem vocabularies in the second half of the fifties.

The verb mechanism is rather sketchy, viz.:

predicate + n + p --> | copula + n + p > (quality) (circumstance) /
| verb + n + p > (object) (circumstance)

Verb, which I shall not write out, has only the present tense, including a differentiation between German strong and weak verbs. In a later stage

other tenses will be added, together with the progressive form and the auxiliary verbs, making use of inversions in the same way as in the sub-stantive.

Relsentence is rather complete:

relsentence + n + p + g -->

(,) relpron + n + g + 1 relpredicate + n + p /
(,) relpron + n + g + 4 relphrase /
(,) preposition + c relpron + n + g + c relphrase/
(,) preposition + c relpron + n + g + 2
noun + n' + g' + c + ending1 relphrase /
(,) relpron + n + g + 2 noun + n' + g' + 1 + ending1
relpredicate + n' + 3 /
(,) relpron + n + g + 2 noun + n' + g' + 4 + ending1
relphrase

Instances of all these are, respectively:

, who sees me
, whom I see
, to whom I go
, to whose house I go
, whose dog smells me
, whose dog I smell

The difference between predicate and relpredicate is one of word order in German. Just compare the rule for predicate with

relpredicate + n + p -->

< (adverb) verb + n + p | (object) (circumstance) > /
< (adverb) copula + n + p | (quality) (circumstance) >

The mysterious "circumstance" is either an indication of when or where the action of the sentence takes place, or a preposition construction like "to me".

If any matter is not immediately clear to the reader then some perusal of the program will solve all doubts. The first part of the program is a set of string handling routines, including a reading procedure, written in machine code. It has a variable number of parameters, but at least four, two arrays and two integers that serve as pointers are always required. The input is on FLEXOWRITER tape. The heptads are packed three in a word. Separator is either a period or a comma. The last word of the packed string is given a negative sign. The integer assigned to the name of the string that is read in, is the location of its first packed word. The other array serves as a list of all known words in order to enable the procedure lees1 to assign the same integer to identical strings. Readn can have at most 27 string names as parameters, a limitation imposed by the running system used. It seems not worthwile to describe the program in any more detail, as the linguist will not be interested and the ALGOList can find out for himself. It should be kept in mind this is a rather preliminary version of the grammar that will be published in a larger and more polished version later. The presented grammar is too small to have more than demonstrational value for the techniques used.

```
begin   comment vertaler ENGELS ---> DUITS, K. Koster, R826 ;
integer halt, vertaal, output, commentaar, n1, l1, l2, l3, l4, x, w, as1, k;
boolean b, q1, q2, op, o, g;
integer array W[0 : 600], T[0 : 1500], I[0 : 50], O[0 : 500], GK[0 : 40];

boolean procedure af; af:= op ∧ x ≥ n1;

procedure PP(i); integer i; PU7BIT(i);

procedure OUTPUT;
begin   integer c, d, e;
        for c:= 0 step 1 until w - 1 do
        begin   d:= O[c]; if d < 0 then
                begin   if d < -30767 then c:= d + 32767 else
                        if d < -256 then print(d) else
                        begin   if d < -127 then
                                begin P(26); d:= d + 128 end;
                                for e:= 1 step 1 until -d do P(16)
                        end
                end             else
                begin   e:= d : 128; P(d - 128 × e); if e = 0 then goto E;
                        d:= e : 128; P(e - 128 × d); if d ≠ 0 then P(d); E:
                end
        end; w:= 0
end;

procedure PRINT(i, j); value i, j; integer i, j;
begin   integer c, d, e;
SC:     if j > 127 then begin PP(26); j:= j - 128; goto SC end;
        for j:= j step -1 until 1 do PP(16);
PC:     c:= T[i]; d:= abs(c);
        e:= d : 128; PP(d - 128 × e); if e = 0 then goto PE;
        d:= e : 128; PP(e - 128 × d); if d ≠ 0 then PP(d); PE:
        if c > 0 then begin i:= i + 1; goto PC end
end;

procedure TYPE(i, j); value i, j; integer i, j;
begin   integer c, d, e;
SC:     if j > 127 then begin P(26); j:= j - 128; goto SC end;
        for j:= j step -1 until 1 do P(16);
PC:     c:= T[i]; d:= abs(c);
        e:= d : 128; P(d - 128 × e); if e = 0 then goto PE;
        d:= e : 128; P(e - 128 × d); if d ≠ 0 then P(d); PE:
        if c > 0 then begin i:= i + 1; goto PC end
end;

procedure PONS(i, j); value i, j; integer i, j;
begin   integer a; O[w]:= -j; w:= w + 1;
        if i ≥ 0 then
        begin
        N:      a:= T[i]; O[w]:= abs(a); w:= w + 1; if a > 0 then
                begin   i:= i + 1; goto N end
        end
end;
```

```
procedure P(i); integer i;
begin   switch PS:= p1, p2, p4, p7, p8, pe, pe, pe, psp, p3, p5, p6, p9,
                       pnl, pe, pe, p0, pt, pv, pw, pz, pe, pe, pe,
                       pe, ps, pu, px, py, pe, pe, ptb;
        switch PSS:= pmin, pl, pn, po, pr, pe, pe, pe, pj, pk, pm,
                       pp, pq, pcom, pe, pe, pa, pb, pd, pg, ph, ppnt,
                       pe, pe, pplus, pc, pee, pf, pi, pe, pe, pe;
        procedure P(u); string u; begin PRINTTEXT(u); goto pe end;
        goto if i = 0 then pe else if i > 63 then PSS[i : 2 - 31] else PS[i : 2 + 1];
        p1: P(⊀1⊁); p2: P(⊀2⊁); p3: P(⊀3⊁); p4: P(⊀4⊁); p5: P(⊀5⊁);
        p6: P(⊀6⊁); p7: P(⊀7⊁); p8: P(⊀8⊁); p9: P(⊀9⊁);
        p0: P(⊀0⊁); pa: P(⊀a⊁); pb: P(⊀b⊁); pc: P(⊀c⊁); pd: P(⊀d⊁);
        pee: P(⊀e⊁); pf: P(⊀f⊁); pg: P(⊀g⊁); ph: P(⊀h⊁);
        pi: P(⊀i⊁); pj: P(⊀j⊁); pk: P(⊀k⊁); pl: P(⊀l⊁); pm: P(⊀m⊁);
        pn: P(⊀n⊁); po: P(⊀o⊁); pp: P(⊀p⊁); pq: P(⊀q⊁);
        pr: P(⊀r⊁); ps: P(⊀s⊁); pt: P(⊀t⊁); pu: P(⊀u⊁); pv: P(⊀v⊁);
        pw: P(⊀w⊁); px: P(⊀x⊁); py: P(⊀y⊁); pz: P(⊀z⊁);
        psp: P(⊀ ⊁); ptb: P(⊀     ⊁); pmin: P(⊀-⊁); pcom: P(⊀,⊁);
        pplus: P(⊀+⊁); ppnt: P(⊀.⊁); pnl: P(⊀
        ⊁); pe:
end;

procedure match(i, j); value i; integer i, j;
begin   if af then begin b:= false; goto ematch end;
        if x = n1 then lees1;
        b:= I[x] = i; if b then
        begin   x:= x + 1; PONS(j, 1);
                if o then PRINT(i, 128 + as1)
        end; ematch:
end;

boolean procedure equal(i, p); value i, p; integer i, p;
begin   M: if T[i] ≠ T[p] then begin equal:= false; goto E end;
        if T[i] > 0 then
        begin   i:= i + 1; p:= p + 1; goto M end;
        equal:= true; E:
end;

procedure lees1;
begin   integer i, q; op:= leesn(I, T, n1, k, q);
        for i:= l2 - 1 step -1 until l1 do
        begin   if equal(q, W[i]) then goto EL end;
        NLCR; NLCR; PRINTTEXT(⊀ UNKNOWN - ⊁);
LTP:    TYPE(q, 1); if ¬ op then
        begin   op:= leesn(I, T, n1, k, q); goto LTP end;
        q1:= true; x:= n1; goto EE;
EL:     I[x]:= W[i]; TYPE(q, 1); EE:
end;
```

```
procedure Q(i); integer i;
begin  if o then PRINT(GK[i], 128 + as1);
       as1:= as1 + 1
end;

procedure UQ;
begin  as1:= as1 - 1; if w > 14 then 14:= w end;

procedure invopen; PONS(-1, 258);

procedure invmiddle; PONS(-1, 257);

procedure invend;
begin  integer t, i, j; i:= w - 1;
AA:    t:= O[i]; if t = -257 then O[i]:= w - 32767
       else begin i:= i - 1; goto AA end;
       j:= i - 1;
BA:    t:= O[j]; if t = -258 then O[j]:= i - 32767
       else begin j:= j - 1; goto BA end;
       O[w]:= j -32767; w:= w + 1
end;

boolean procedure leesn;
begin  leesn:= false; KODE(⊀
dn+128ds0
dpzr0ze3 dpzf0ze1 dpzk0ze2 dpzw24ze2 dpzn20zk0
dida0ze0
2b19x1 2b9x0b 2s0x0b 6s0w0 2b19x1 2b5x0b 0s0x0b 6s1w0
2b19x1 2b11x0b 2s0x0b 6s2w0 2b19x1 2b7x0b 0s0x0b 6s3w0
2b19x1 0b13a 2s2w0 u1b18x1z y2t0zr0a 6b4w0 2b0x0b 6s0x0b
6t0f00 2b4w0 0b2a 2t19e0a
da0zf0di
2y1xpz n0la16az 0la16a y2t0f0a 6t1k01 2s0a n6t1n02 y1p7ss
n6t0n02 y1p7ss n6t0n02 y1p7ss n6t0k01 1p13ss y5pss 2b3w0
6s0x0b 0b1a 6b3w0 2s1a 4s2w0 n2t5f0a 4s0w0 2s2w0
2b1w0 6s0x0b 0b1a 6b1w0 2t8x0e
da0zk0di
2y1xpz y2t0k0a u0la26az y2t0k0a u0la62az y2t0k0a u1a121ap y2t0k0a
u0la91az y2a1a y2t18k0a u0la107az n2t9x0z 2b4w0 0b2a u1b18x1z
n7y29c0 2a0az 6a5w0 2t9x0z 6t0k01 n1p7ss n0x1a 2t10x0z
da0zr0di
2s0w0 2b19x1 2b9x0b 6s0x0b 2b19x1 2b11x0b 2s2w0 6s0x0b
2s5w0 2b18x1 6s32764x0b 2s0a 6s32767x0b dq+13 dso⊁);
       comment leesn becomes true iff the last separator is a period
end;


x:= n1:= k:= 0;
leesn(W, T, n1, k, halt, vertaal, output, commentaar);
TYPE(commentaar, 130);
leesn(W, T, n1, k, as1); q2:= equal(as1, output);
leesn(W, T, n1, k, as1); g:= equal(as1, halt); n1:= 0;
```

```
begin   comment grammar proper; integer
        denn, und, oder, auch, d, ein, en, er, e, es, immer, oft,
        nie, nicht, hier, dort, ueberall, wo, wenn, von, nach,
        fuer, schoen, klein, schnell, gluecklich, ie, em, zeig,
        sen, ich, wir, du, sie, mir, uns, dir, ihm, ihr, ihnen, mich,
        dich, ihn, n, mann, hund, maenner, hunde, frau, frauen,
        stadt, staedte, kind, kinder, haus, haeuser, geh, seh,
        frag, geb, gib, ess, isz, st, t, fast nicht, sieh, sehr,
        ziemlich, kenn, bin, bist, ist, sind, seid, ha, hab, mit,
        garten, wohn, aber, as,
        for, and, or, comma, too, the, a, an, always, often, never,
        nearly, fast, hardly, not, here, there, everywhere, where,
        when, to, from, after, who, in, beautiful, small, happy, see,
        that, whose, whom, i, we, you, he, they, she, it, me, us,
        him, her, them, man, dog, men, dogs, woman, women, city,
        cities, child, house, children, houses, go, goes, garden,
        sees, ask, asks, know, knows, show, shows, give, gives, eat,
        eats, very, rather, are, am, is, in, have, has, live, lives,
        into, with, but;

        procedure sentence;
        begin   as1:= 0; Q(0);
                basicsentence; if af then goto E;
                nextsentence; E:
        end;

        procedure nextsentence;
        begin   connective; if b then sentence end;

        procedure connective;
        begin   integer z; z:= w; Q(1);
                connecti1; if ¬b then connecti2;
                if ¬b then w:= z; E: UQ
        end;

        procedure connecti1;
        begin   integer y, z; y:= x; z:= w;
                match(comma, comma);
                match(for, denn); if b then goto E;
                match(but, aber); if b then goto E;
                x:= y; w:= z; E:
        end;

        procedure connector;
        begin   integer y, z; y:= x; z:= w; Q(26);
                match(comma, comma);
                if ¬b then connective2;
                if ¬b then begin x:= y; w:= z end; UQ
        end;
```

```
procedure connecti2;
begin   match(and, und); if b then goto E;
        match(or, oder); E:
end;

procedure basicsentence;
begin   integer y, z, n, p; y:= x; z:= w;
        subject(n, p); if ¬ b then goto A; invopen;
T:      predicate(n, p); if ¬b then goto F; goto E;
A:      circumstance; if ¬b then goto F;
        invopen; subject(n, p); if b then goto T;
F:      x:= y; w:= z; E:
end;

procedure subject(n, p); integer n, p;
begin   integer y, z, g; y:= x; z:= w; Q(3);
        invopen; nounphrase(n, p, g, 1); if ¬b then goto F;
        invmiddle; match(too, auch); invend;
        b:= true; goto E;
F:      x:= y; w:= z; E: UQ
end;

procedure object;
begin   integer y, z, n, p, g; y:= x; z:= w; Q(4);
        invopen; nounphrase(n, p, g, 4); if ¬b then goto F;
        invmiddle; match(too, auch); invend;
        nextobject; b:= true; goto E;
F:      x:= y; w:= z; E: UQ
end;

procedure nextobject;
begin   integer y, z; y:= x; z:= w;
        connector; if b then object;
        if ¬b then begin x:= y; w:= z end
end;

procedure nounphrase(n, p, g, c); integer n, p, g, c;
begin   integer y, z; y:= x; z:= w; Q(5);
        perspron(n, p, g, c); if ¬b then goto A;
T:      relsentence(n, p, g); if ¬b then circumstance;
        b:= true; goto E;
A:      nounpart(n, g, c); if ¬b then goto F; p:= 3; goto T;
F:      x:= y; w:= z; E: UQ
end;

procedure predicate(n, p); integer n, p;
begin   integer y, z; y:= x; z:= w; Q(7);
        invmiddle; copula(n, p); if ¬b then goto A;
        invend; quality; circumstance; b:= true; goto E;
A:      verb(n, p); if ¬b then goto F; invend;
        object; circumstance; b:= true; goto E;
F:      x:= y; w:= z; E: UQ
end;
```

```
procedure quality;
begin   integer n, p, g, y, z; y:= x; z:= w; Q(2);
        nounphrase(n, p, g, 1); if b then goto E;
        adverb; modifier; stemofadjective;
        if ¬b then begin x:= y; w:= z end; E:
end;

procedure copula(n, p); integer n, p;
begin   integer y, z; y:= x; z:= w; Q(6);
        match(if n = 2 ∨ p = 2 then are else if p = 1 then am else is,
        if n = 1 then(if p = 1 then bin else if p = 2 then bist else ist)
                else (if p = 2 then seid else sind)); if b then goto E;
        x:= y; w:= z; E: UQ
end;

procedure nounpart(n, g, c); integer n, g, c;
begin   integer y, z; y:= x; z:= w; Q(8);
        match(the, d); if ¬b then goto A;
        invopen; noun(n, g, c, ending1); if ¬b then goto F;
        invmiddle; endingofder(n, g, c); invend; goto E;
A:      match(a, ein); if ¬b then match(an, ein); if ¬b then goto B;
        invopen; noun(n, g, c, ending2); if ¬b then goto F;
        invmiddle; endingofein(n, g, c); invend; goto E;
B:      noun(n, g, c, ending3); if b then goto E;
F:      x:= y; w:= z; E: UQ
end;

procedure noun(n, g, c, uitgang); integer n, g, c; procedure uitgang;
begin   integer y, z; y:= x; z:= w; Q(9);
        modifier; stemofadjective; if ¬b then goto A;
        invopen; noun(n, g, c, uitgang); if ¬b then goto F;
        invmiddle; uitgang(n, g, c); invend; goto E;
A:      subst(n, g, c); if b then goto E;
F:      x:= y; w:= z; E: UQ
end;

procedure modifier;
begin   integer z; z:= w; Q(22);
        match(very, sehr); if b then goto E;
        match(rather, ziemlich); if b then goto E;
        w:= z; E: UQ
end;

procedure relphrase;
begin   integer y, z, n, p; y:= x; z:= w; Q(10);
        subject(n, p); if ¬b then goto F;
        relpredicate(n, p); if b then goto E;
F:      x:= y; w:= z; E: UQ
end;
```

```
procedure relpredicate(n, p); integer n, p;
begin   integer y, z; y:= x; z:= w; Q(11);
        invopen; adverb; copula(n, p); if ¬b then goto A;
        invmiddle; quality; prepclause;
B:      invend; circumstance; b:= true; goto E;
A:      verb(n, p); if ¬b then goto F;
        invmiddle; object; goto B;
F:      x:= y; w:= z; E: UQ
end;

procedure relsentence(n, p, g); integer n, p, g;
begin   integer y, z, c, n1, g1; y:= x; z:= w; Q(12);
        match(comma, comma);
        relpron(n, g, 1); if ¬b then goto A;
        relpredicate(n, p); goto G;
A:      relpron(n, g, 4); if ¬b then goto B;
D:      relphrase; goto G;
B:      preposition(c); if ¬b then goto C;
        relpron(n, g, c); if ¬b then goto T; goto D;
G:      if b then goto E; goto F;
C:      relpron(n, g, 2); if ¬b then goto F;
        noun(n1, g1, 1, ending1);
        if b then relpredicate(n1, 3); if b then goto E;
        noun(n1, g1, 4, ending1); if ¬b then goto F;
        relphrase; goto G;
T:      relpron(n, g, 2); if ¬b then goto F;
        noun(n1, g1, c, ending1); if b then goto D;
F:      x:= y; w:= z; E: UQ
end;

procedure ending1(n, g, c); integer n, g, c;
PONS(if (c = 4 ∧ g = 1) ∨ c = 3 ∨ c = 2 ∨ n = 2 then en else e, 0);

procedure ending2(n, g, c); integer n, g, c;
PONS(if g = 1 then (if c = 1 then er else if c = 2 then es else en)
        else if c = 2 ∨ c = 3 then en else if g = 2 then e else es, 0);

procedure ending3(n, g, c); integer n, g, c;
begin   switch C:= qe, qer, qen, qe, qer, qen, qem, qen,
                 qe, qer, qer, qe, qes, qen, qem, qes;
        goto C[if n = 1 then 4 × g + c else c];
        qe: PONS(e, 0); goto E; qer: PONS(er, 0); goto E;
        qen: PONS(en, 0); goto E; qem: PONS(em, 0); goto E;
        qes: PONS(es, 0); E:
end;

procedure adverb;
begin   adverb1; if ¬b then adverb2 end;
```

```
procedure adverb1;
begin   match(always, immer); if b then goto E;
        match(often, oft); if b then goto E;
        match(never, nie); E:
end;

procedure adverb2;
begin   match(nearly, fast); if b then goto E;
        match(hardly, fast nicht); E:
end;

procedure adverb3;
begin   match(here, hier); if b then goto E;
        match(there, dort); if b then goto E;
        match(everywhere, ueberall); E:
end;

procedure temporalclause;
begin   integer y, z; y:= x; z:= w; Q(23);
        adverb1; if b then goto A;
        clause1; if b then goto E; goto F;
A:      clause1; match(comma, comma); b:= true; goto E;
F:      x:= y; w:= z; E: UQ
end;

procedure clause1;
begin   integer y, z; y:= x; z:= w;
        match(when, wenn); if ¬b then goto E;
        relphrase; if ¬b then begin x:= y; w:= z end; E:
end;

procedure spatialclause;
begin   integer y, z; y:= x; z:= w; Q(24);
        adverb3; if b then goto A;
        clause3; if b then goto E; goto F;
A:      clause3; match(comma, comma); b:= true; goto E;
F:      x:= y; w:= z; E: UQ
end;

procedure clause3;
begin   integer y, z; y:= x; z:= w;
        match(where, wo); if ¬b then goto E;
        relphrase; if ¬b then begin x:= y; w:= z end; E:
end;

procedure prepclause;
begin   integer y, z, n, p, g, c; y:= x; z:= w; Q(25);
        preposition(c); if b then nounphrase(n, p, g, c);
        if ¬b then begin x:= y; w:= z end; UQ
end;
```

```
procedure circumstance;
begin   integer y, z; z:= w; y:= x; Q(14);
        prepclause; if b then goto E;
        temporalclause; if b then goto E;
        spatialclause;
        if ¬b then begin x:= y; w:= z end; E: UQ
end;

procedure preposition(c); integer c;
begin   integer z; z:= w; Q(15); c:= 3;
        match(in, in); if b then goto E;
        match(with, mit); if b then goto E;
        match(to, nach); if b then goto E;
        match(from, von); if b then goto E;
        match(after, nach); if b then goto E; c:= 4;
        match(into, in); if b then goto E;
        match(for, fuer);
        if ¬b then w:= z; E: UQ
end;

procedure stemofadjective;
begin   integer z; z:= w; Q(16);
        match(beautiful, schoen); if b then goto E;
        match(small, klein); if b then goto E;
        match(fast, schnell); if b then goto E;
        match(happy, gluecklich);
        if ¬b then w:= z; E: UQ
end;

procedure relpron(n, g, c); integer n, g, c;
begin   integer z; switch CASE:= C1, C2, C3, C4;
        z:= w; Q(17); goto CASE[c];

C1:     match(who, d); if b then goto A;
        match(that, d); if b then goto A;
        goto F;
C2:     match(whose, d); if b then goto A;
        goto F;
C3:     match(whom, d); if b then goto A;
        goto F;
C4:     match(whom, d); if b then goto A;
        match(that, d); if b then goto A;
        goto F;

A:      endingofder(n, g, c);
        if n = 1 then
        begin if c = 2 then PONS(if g = 2 then en else sen, 0) end
                else if c = 2 ∨ c = 3 then PONS(en, 0); goto E;
F:      w:= z; E: UQ
end;
```

```
procedure endingofder(n, g, c); integer n, g, c;
begin    switch S:= die, der, den, die, der, des, dem, den,
                   die, der, der, die, das, des, dem, das;
         goto S[if n = 2 then c else 4 × g + c];
die:     PONS(ie, 0); goto E; der:         PONS(er, 0); goto E;
des:     PONS(es, 0); goto E; das:         PONS(as, 0); goto E;
dem:     PONS(em, 0); goto E; den:         PONS(en, 0); E:
end;

procedure endingofein(n, g, c); integer n, g, c;
if ¬((g = 1 ∧ c = 1) ∨ (g ≠ 1 ∧ (c = 1 ∨ c = 4)))
         then endingofder(n, g, c) else if g = 2 then PONS(e, 0);

procedure perspron(n, p, g, c); integer n, p, g, c;
begin    integer z; switch CASE:= C1, C2, C3, C4;
         z:= w; Q(18); n:= 1; p:= 1; g:= 1;
         goto CASE[c];

C1:      match(i, ich); if b then goto E; n:= 2;
         match(we, wir); if b then goto E; n:= 1; p:= 2;
         match(you, du); if b then goto E; p:= 3;
         match(he, er); if b then goto E; n:= 2;
         match(they, sie); if b then goto E; g:= 2; n:= 1;
         match(she, sie); if b then goto E; g:= 3;
         match(it, es); goto F;

C2:      b:= false; goto F;

C3:      match(me, mir); if b then goto E; n:= 2;
         match(us, uns); if b then goto E; n:= 1; p:= 2;
         match(you, dir); if b then goto E; p:= 3;
         match(him, ihm); if b then goto E; g:= 2;
         match(her, ihr); if b then goto E; g:= 3;
         match(it, ihm); if b then goto E; n:= 2; g:= 1;
         match(them, ihnen); goto F;

C4:      match(me, mich); if b then goto E; n:= 2;
         match(us, uns); if b then goto E; n:= 1; p:= 2;
         match(you, dich); if b then goto E; p:= 3;
         match(him, ihn); if b then goto E; g:= 2;
         match(her, sie); if b then goto E; g:= 3;
         match(it, es); if b then goto E; n:= 2; g:= 1;
         match(them, sie);

F:       if ¬b then w:= z; E: UQ
end;
```

```
procedure subst(n1, g, c); integer n1, g, c;
begin   integer y, z, t; y:= x; z:= w; Q(19);
        g:= 1; n1:= 1;
        match(man, mann); if b then goto A;
        match(dog, hund); if b then goto A;
        match(garden, garten); if b then goto A;
        n1:= 2;
        match(men, maenner); if b then goto A;
        match(dogs, hunde); if b then goto A;
        g:= 2; n1:= 1;
        match(woman, frau); if b then goto E;
        match(city, stadt); if b then goto E;
        n1:= 2;
        match(women, frauen); if b then goto E;
        match(cities, staedte); if b then goto E;
        g:= 3; n1:= 1;
        match(child, kind); if b then goto A;
        match(house, haus); if b then goto A;
        n1:= 2;
        match(children, kinder); if b then goto A;
        match(houses, haeuser); if ¬b then goto F;
A:      if n1 = 1 then
        begin   if c = 2 then t:= es else
                if c = 3 then t:= e else goto E
        end       else if c = 3 then t:= n else goto E;
        PONS(t, 0); goto E;
F:      x:= y; w:= z; E: UQ
end;

procedure verb(n, p); integer n, p;
begin   integer y, z;
        integer procedure dkeuze(a, b); integer a, b;
        dkeuze:= if n = 1 then (if p = 1 then a else b)
                           else (if p = 2 then b else a);
        integer procedure ekeuze(a, b); integer a, b;
        ekeuze:= if n = 1 ∧ p = 3 then b else a;
        y:= x; z:= w; Q(20);
        match(ekeuze(go, goes), geh); if b then goto E;
        match(ekeuze(see, sees), dkeuze(seh, sieh)); if b then goto E;
        match(ekeuze(ask, asks), frag); if b then goto E;
        match(ekeuze(give, gives), dkeuze(geb, gib)); if b then goto E;
        match(ekeuze(have, has), if n = 2 ∨ p = 1 then hab else ha);
        if b then goto E; match(ekeuze(live, lives), wohn); if b then goto E;
        match(ekeuze(eat, eats), dkeuze(ess, isz)); if b then goto E;
        match(ekeuze(know, knows), kenn); if b then goto E; goto F;
E:      if p = 2 then PONS(if n = 1 then st else t, 0) else
        PONS(if n = 1 then (if p = 1 then e else t) else en, 0); goto R;
F:      x:= y; w:= z; R: UQ
end;
```

```
           begin   integer i1, q;
                   for i1:= 0 step 1 until 26 do
                   begin   b:= leesn(W, T, n1, k, q); GK[i1]:= q end;
                   i1:= n1; b:= leesn(W, T, n1, k,
                   denn, und, oder, auch, d, ein, en, er, e, es, immer,
                   oft, nie, nicht, hier, dort, ueberall, wo, wenn, von,
                   nach, fuer, schoen, klein, schnell, gluecklich);
                   b:= leesn(W, T, n1, k,
                   ie, em, zeig, sen, ich, wir, du, sie, mir, uns,
                   dir, ihm, ihr, ihnen, mich, dich, ihn, n, mann,
                   hund, maenner, hunde, frau, frauen, stadt, staedte);
                   b:= leesn(W, T, n1, k,
                   kind, kinder, haus, haeuser, geh, seh, frag, geb,
                   gib, ess, isz, st, t, fast nicht, sieh, sehr,
                   ziemlich, kenn, bin, bist, ist, sind, seid, as);
                   b:= leesn(W, T, n1, k,
                   hab, ha, garten, wohn, mit, aber,
                   have, has, garden, live, lives, into, with, but);
                   b:= leesn(W, T, n1, k,
                   for, and, or, comma, too, the, a, an, always, often, never,
                   nearly, fast, hardly, not, here, there, everywhere,
                   where, when, to, from, after, who, beautiful);
                   b:= leesn(W, T, n1, k,
                   small, happy, see, that, whose, whom, i, we, you, he,
                   they, she, it, me, us, him, her, them, man,
                   dog, men, dogs, woman, women, city, cities);
                   b:= leesn(W, T, n1, k,
                   child, house, children, houses, go, goes, sees, ask, asks,
                   know, knows, show, shows, give, gives, eat, eats,
                   very, rather, are, am, is, in);

                   12:= n1; 13:= k; o:= q2; n1:= 0; if ¬g then stop;


           S:      if g then stop; NLCR;
                   if o then begin RUNOUT; PUNLCR end;
                   x:= w:= 14:= 0; if q2 ∨ ¬o then
                   begin   leesn(I, T, n1, k, q); if equal(q, halt) then goto einde;
                           n1:= x:= 0; op:= false
                   end;
                   q1:= false; sentence; if b then PRINTTEXT(⋠
ja         ⋡)      else begin PRINTTEXT(⋠
nee        ⋡);     w:= 14 end;
                   OUTPUT; o:= (¬q1 ∧ ¬b ∧ ¬o) ∨ q2; k:= 13;
                   if ¬b ∧ ¬op then
                   begin A: op:= leesn(I, T, n1, k, q); if ¬op then goto A end;
                   goto S;
einde:     end; STOPCODE; RUNOUT
end
end
```

5.2 Some test results

The program as reproduced was run on the X1 of the Mathematical Centre, Amsterdam. The translated program occupied about 6500 of the 10 000 memory places available, the arrays another 2700; all told, the available room for the program stack will not have been over 500 places. Some 1000 places in the arrays could still be gained by lowering the array bounds, accomodating some 80 structures of the form

match(a1, a2); if ¬b then goto E;

By some slight alterations still more space could be made available for enlarging the grammar, but it is clear that the 12K memory cannot accommodate a grammar over 50 percent larger.
With the present grammar, a mean translating time of about 4.3 seconds per word is reached. Of this time, more than 2 seconds is spent in the procedure lees1, searching the vocabulary for the word just read in order to determine an integer to represent the word. As this is done in ALGOL (and by no means in the most practical way - the words are not alphabetized), it is painfully slow. This is readily understandable, since the addition of two integers requires 3 msec. in X1 ALGOL, and another 3 msec. is required to store the result. On a computer that is a hundred times faster, speed will not be the limiting factor.

The input consists of some sentences that display various difficulties: "and" used as a connective and as connector; the mildly ridiculous effect of translating idiom literally; the various word orders in German; and, finally, a grammatical sentence that is analysed incorrectly.
When this sentence fails to be translated, the program tries again, this time punching the name of every category as it is tried. It fails again, of course, and skips to the next sentence, instead of which it finds an indication to stop.

5.3 Input II and output IV

stop, translate, output,

k koster-testsentences.

no output of structure, no stopping between sentences.

sentence, connective, quality, subject, object, nounphrase, copula, predicate, nounpart, noun, relphrase, relpredicate, relsentence, adverb, circumstance, preposition, adjective, relpron, perspron, subst, verb, unused, modifier, temporal clause, spatial clause, preposition clause, connector.

denn, und, oder, auch, d, ein, en, er, e, es, immer,
oft, nie, nicht, hier, dort, ueberall, wo, wenn, von,
nach, fuer, schoen, klein, schnell, gluecklich.
ie, em, zeig, sen, ich, wir, du, sie, mir, uns,
dir, ihm, ihr, ihnen, mich, dich, ihn, n, mann,
hund, maenner, hunde, frau, frauen, stadt, staedte.
kind, kinder, haus, haeuser, geh, seh, frag, geb,
gib, ess, isz, st, t, fast nicht, sieh, sehr,
ziemlich, kenn, bin, bist, ist, sind, seid, as.
hab, ha, garten, wohn, mit, aber,
have, has, garden, live, lives, into, with, but.
for, and, or, comma, too, the, a, an, always, often, never,
nearly, fast, hardly, not, here, there, everywhere,
where, when, to, from, after, who, beautiful.
small, happy, see, that, whose, whom, i, we, you, he,
they, she, it, me, us, him, her, them, man,
dog, men, dogs, woman, women, city, cities.
child, house, children, houses, go, goes, sees, ask, asks,
know, knows, show, shows, give, gives, eat, eats,
very, rather, are, am, is, in.

translate.
a, man, sees, a, small, house, and, in, the, house, he, sees,
a, woman, and, a, child.

translate.
the, man, goes, to, the, house, for, he, sees, a, dog, too,
and, he, knows, dogs, that, eat, children.

translate.
the, dog, is, for, the, man, who, is, very, happy, when, the, woman,
gives, the, dog, comma, for, the, man, has, a, small, child, too.

translate.
i, know, a, happy, man, when, i, see, him.

translate.
the, man, that, sees, a, dog, sees, the, dog, from, the, house.

translate.
the, woman, in, whose, house, i, live, has, a, small, beautiful, garden, too.

translate.
a, small, garden, is, a, garden, that, is, rather, small.

translate.
i, live, here, and, she, lives, there, but, he, lives, everywhere,
where, she, lives.

translate.
i, go, into, the, house, when, i, see, him, comma, for, i, know, him.

translate.
when, i, see, him, with, a, dog, i, go, into, the, house.

translate.
i, see, you, and, you, see, me.

stop.

k koster-testsentences

	a man sees a small house and in the house he sees a woman and a child
ja	ein mann sieht ein kleines haus und in dem hause sieht er eine frau und ein kind
	the man goes to the house for he sees a dog too and he knows dogs that eat children
ja	der mann geht nach dem hause denn er sieht auch einen hund und er kennt hunde die kinder essen
	the dog is for the man who is very happy when the woman gives the dog comma for the man has a small child too
ja	der hund ist fuer den mann der sehr gluecklich ist wenn die frau den hund gibt comma denn der mann hat auch ein kleines kind
	i know a happy man when i see him
ja	ich kenne einen gluecklichen mann wenn ich ihn sehe
	the man that sees a dog sees the dog from the house
ja	der mann der einen hund sieht sieht den hund von dem hause
	the woman in whose house i live has a small beautiful garden too
ja	die frau in deren hause ich wohne hat auch einen kleinen schoenen garten
	a small garden is a garden that is rather small
ja	ein kleiner garten ist ein garten der ziemlich klein ist
	i live here and she lives there but he lives everywhere where she lives
ja	ich wohne hier und sie wohnt dort aber er wohnt ueberall wo sie wohnt
	i go into the house when i see him comma for i know him
ja	ich gehe in das haus wenn ich ihn sehe comma denn ich kenne ihn
	when i see him with a dog i go into the house
ja	wenn ich ihn mit einem hunde sehe gehe ich in das haus
	i see you and you see
neê	ich sehe dich und dich

```
sentence
 subject
  nounphrase
   perspron
    i
   relsentence
    relpron
    relpron
    preposition
    relpron
   circumstance
    preposition clause
     preposition
    temporal clause
    spatial clause
 predicate
  copula
  verb
   see
  object
   nounphrase
    perspron
     you
    relsentence
     relpron
     relpron
     preposition
     relpron
    circumstance
     preposition clause
      preposition
     temporal clause
     spatial clause
   connector
    connective
     and
   object
    nounphrase
     perspron
      you
     relsentence
      relpron
      relpron
      preposition
      relpron
     circumstance
      preposition clause
       preposition
      temporal clause
      spatial clause
    connector
     connective
  circumstance
   preposition clause
    preposition
   temporal clause
   spatial clause
 connective
```

6. References

The predictive analysis technique seems to have originated with RHODES in 1958; in 1959 it was in the proceedings of every self-respecting symposium on mechanical linguistics. This author was introduced to it in august '64 by

> Howard H. METCALFE , A parameterized compiler based on mechanical linguistics, march 64, Planning Research Corporation, Los Angeles, Calif. Washington D.C.

At the moment, both the predictive analysis technique and the matching technique, which we rejected at the beginning of (2.0) as being not simple enough, have been perfected to such an extent that one can say that the problem of analysing a sentence of a CF language is completely solved. To give an interesting example: GREIBACH presents an analyser that is ideal in the sense that it gives all possible analyses of the sentence, by creating a number of simultaneous analysis automata (stacks). The program makes use of a special normal form for the CF rules that makes it very fast at the cost of memory space.
The idea of affixes as used here, originated with MEERTENS in '62, while he and the author were working on the generation of sentences with the aid of machinecode. As a result of this work a fair sized affix grammar of English was presented in a mimeographed form to some linguists and mathematicians of the late prof. BETH's colloquium on Machines and Language. From a COMIT manual the author knows that some form of affixes is known there, but he does not know to what use it is put.
Some primitive form of simultaneous grammar can be found in many articles on, for instance, syntax based ALGOL compilers, published in ACM in the course of '63 and '64.

This small publication hopes to demonstrate that a linguist with a working knowledge of ALGOL has no need for a large team of collaborators and extensive financial backing to perform experiments that show the linguistic issues involved much more clearly than mere discussion of results achieved elsewhere.