EXERCISES IN COMPUTATIONAL LINGUISTICS

EXERCISES IN COMPUTATIONAL LINGUISTICS

AKADEMISCH PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR IN DE WISKUNDE EN NATUURWETENSCHAPPEN AAN DE UNIVERSITEIT VAN AMSTERDAM, OP GEZAG VAN DE RECTOR MAGNIFICUS, MR. A. D. BELINFANTE, HOOGLERAAR IN DE FACULTEIT DER RECHTSGELEERDHEID, IN HET OPENBAAR TE VERDEDIGEN IN DE AULA DER UNIVERSITEIT (TIJDELIJK IN DE LUTHERSE KERK, INGANG SINGEL 411, HOEK SPUI) OP WOENSDAG 21 JANUARI 1970 DES NAMIDDAGS TE 4 UUR

DOOR

HUGO BRANDT CORSTIUS
GEBOREN TE EINDHOVEN

1970

MATHEMATISCH CENTRUM - AMSTERDAM

Promotor : Prof.dr.ir. A. van Wijngaarden

Co-referent : Prof.dr. F.E.J. Kruseman Aretz

aan mijn ouders

TABLE OF CONTENTS

Introduction

Not the contents of the following chapters are important, but the method used in them. For want of a better term, one could name this method the "constructive" one. If it is pleasant and useful to philosophize, discuss and talk about linguistic problems, it is surely more useful and pleasant to bring linguistic problems to a solution which is so exact that even an electronic computer can be programmed to do the work. This does not mean that the solution has to be the correct one in all possible cases, this often being an unrealizable ideal. What it does mean is that in all possible cases an assertion about the problem must be made. It is in this respect that the electronic computer has importance for the solving of linguistic problems, apart from its consistency and speed: the solution may be wrong but never vague or ambivalent. In the search for the solution, intuition, language sense or whatever one may call these nonlogical sources of insight, may play a part, but once the solution is reached, these nonlogical aids may not be employed in the application of the solution method proposed. The computer which notably lacks intuition, language sense, and so on, is a guarantee for this condition.

In this paper some computer applications to written language, usually Dutch, are dealt with. The solutions are given in the form of programs in ALGOL 60, a machine-independent programming language, not only for numerical processes, but in general fit to describe complicated processes with absolute rigour. It is to be hoped that this unambiguous language or its successor will become, also outside numerical mathematics, the language of scientific conscience.

In the first chapter the forming of Dutch words is investigated. The concept of the syllable is central here. It has proved possible

to isolate the spelling syllable automatically (1.1). This program for hyphenation is used in the automatic typesetting of texts. The program was applied to a collection of newspaper words. This enabled a quantitative analysis of the three parts of the Dutch spelling syllable to be made (1.2). On this basis a generative grammar is found which enables us to produce pronounceable "Dutch-like" words, e.g., new trade names (1.3).

In the second chapter the construction of compound words is exemplified in the construction of the names of cardinal numbers. For five languages a procedure is written to translate from the digital representation of a number into its word representation (2.1). The reverse way is not difficult (2.2) and with these two procedures mutual translation of cardinal number names in different languages is realized (2.3).

In the third chapter the forming of word groups for Dutch sentences is investigated. The noun phrase serves as an example. A grammar is proposed which is used to isolate maximal noun phrases from Dutch newspaper texts (3.1). The application of the program yields two results: a collection of noun phrases with their tree structures (3.2), and a collection of sentences in which the noun phrases are compressed and which thereby have acquired a simpler structure (3.3).

In the fourth chapter, sentences are investigated. Here, the meaning of those sentences is central. The part of meaning has been a growing one in the preceding three chapters: in the first chapter, the meaning of the words to be hyphenated has relevance only in very rare cases. In the second chapter, the meaning of the words, which has to be invariant during translation, is of course relevant, but its structure (the decimal representation of numbers) makes it possible to describe these meanings completely. In the third

chapter, the meaning of the sentences in which we want to isolate the noun phrases presents many difficulties. We limited ourselves in the fourth chapter to a situation where the meaning of segments of discourse can be operationally defined. The meaning of a word problem in algebra is "understood" if its right solution is found. We investigate word problems leading to quadratic equations. Many problems with words, compound words, word groups and sentences are still awaiting a mechanical treatment. It was also our aim to stimulate others in this direction with the examples in this work.

Chapter 1 SYLLABLES

1.1. Isolation of syllables in Dutch words *

1.1.1. Introduction

The fact that most users of the Dutch language will hyphenate most Dutch words in the same way forms a challenge to teach a computer to do the same. The official rules for the division of Dutch words into spelling syllables, as given in the official word list [1], are not directly applicable. They use concepts like pronounceability and word derivation which are even harder to manage for computers than for men.
In general, two - often conflicting - principles in hyphenation can be discerned:

1. Join the longest pronounceable consonant cluster to the next syllable: avon-turen.
2. Divide according to word formation: avond-uren.

For some words, like buurtje, kwartslagen, loodspet the meaning, and thereby the context, determines the hyphenation positions. Their number is so small that we shall assume that for identification of the spelling syllables we shall only need the letters of the word itself.
A trivial solution for the problem of automatic hyphenation is to store in the memory of the computer a list of all Dutch words with their correct splittings. This is the procedure often advised for English hyphenation. This solution would not suffice in our case, however, as there occur many unpredictable compounds in Dutch text which makes any finite vocabulary too short. A solution which costs considerably less time and memory space is, however, possible. We are concerned solely with the spelling syllable, not the phonetic or phonological syllable. Also, we shall not discuss the question whether the existing rules for hyphenation make sense. We just want to let the computer imitate the usual manner of hyphenation.

*This section is an amended translation of communication MR 67, "Automatisch tellen en scheiden van Nederlandse lettergrepen", Mathematical Centre, 1964.

A related, but simpler, problem than the splitting of Dutch words into syllables is the determination of the number of syllables in a Dutch word. This problem is solved in 1.1.2. In 1.1.3, the division in syllables is treated. We give the analysis of both problems, the solution in the form of an ALGOL 60 program [2], the results obtained, some applications, and remarks on the same problems for other languages.

1.1.2. Counting of syllables

1.1.2.1. Analysis

Given: a Dutch word in computer-readable form. To determine: the number of syllables in the word.
We define:

onevowel:	a, e, i, o, u, y.
onecons:	b, c, d, f, ..., x, z.
oneletter:	onevowel or onecons.
twovowel:	aa, au, ay, ..., ij, uy.
twocons:	qu, ch.
twoletter:	twovowel or twocons.
threevowel:	aai, aau, aay, ..., eui, oei.
vowel:	onevowel, twovowel or threevowel.
consonant:	onecons or twocons.
(generalized) letter:	vowel or consonant.
uncondensed letter:	letter occurring in the left part of a twoletter or threevowel, e.g., a, q, aa, i.
condensing letter:	oneletter occurring as last element of a twoletter or threevowel, e.g., a, u, i, j.

The word to be treated is at first considered as a sequence of oneletters. These are then condensed from left to right into maximal letters. This compression is represented easily in a matrix with a row for each uncondensed letter and a column for each condensing letter. In the matrix, the resulting letter for a given pair of letters is given in case compression is possible. From left to right, the uncondensed letters are, if possible,

combined with their immediate successors. In this way, the five oneletters of the word aaien become, after compression, the three generalized letters aai, e and n. The i was not combined with the e to form ie, as it was already used to condense the twovowel aa. The accidental circumstance that the first two oneletters of each threevowel are always a twovowel makes this procedure possible. The only exception is the threevowel eau (in French borrowings), which makes special measures necessary. To this end, the (e,a)-position in the matrix is given the value -1. The compression matrix for Dutch is:

	a	e	i	u	o	y	j	h
aa			aai	aau		aay		
ee				eeu				
oo			ooi			ooy		
a	aa	ae	ai	au		ay		
e	-1	ee	ei	eu		ey		
i		ie					ij	
u			ui	uu		uy		
o		oe	oi	ou	oo	oy		
eu			eui					
ie				ieu				
oe			oei	oeu		oey		
ou			oui					
q				qu				
c								ch

For internal use in the computer, the letters must have code values attached to them. This code is not essential, but a coding proper to the problem enhances readability and speed of the program. Our code, also chosen with reference to the hyphenation program in 1.1.3, is as follows:

1 aai	17 ij	33 o	49 m
2 aay	18 oi	34 eu	50 n
3 oei	19 oui	35 ie	51 p
4 oey	20 oy	36 oe	52 r
5 ooi	21 ui	37 ou	53 s
6 ooy	22 uy	38 y	54 t
7 aau	23 eau	39 q	55 v
8 ae	24 oeu	40 c	56 w
9 ai	25 uu	41 j	57 x
10 au	26 aa	42 h	58 z
11 ay	27 ee	43 k	59 diaeresis(underlining)
12 eeu	28 oo	44 b	60 qu
13 ei	29 a	45 d	61 ch
14 eui	30 e	46 f	62 apostrophe
15 ey	31 i	47 g	63 case definitions
16 ieu	32 u	48 l	64 hyphen

This code makes it possible to differentiate easily between groups such as "vowels", "syllable delimiters", "condensing letters", and so on.
The y is considered as a vowel, and in a word like royaal condenses the preceding o to the vowel oy, making a correct hyphenation.
The compression matrix is translated into this code as an *integer array compression*, the empty places get value zero.
If one would want to make changes in the compression (e.g., recognize ea as a vowel, or not combine the digraph ch), then this is possible without changing the program itself.
After compression of the oneletters from left to right, the number of syllables is equal to the number of vowels.
The only necessary information is therefore: the vowel quality of some generalized letters and the compression matrix.
Example: quadraat contains eight oneletters. After compression, it contains six letters: qu, a, d, r, aa and t, of which a and aa are vowels. Therefore, the number of syllables is two.

1.1.2.2. ALGOL program

In order to make the program independent of the accidental coding system used in our installation, the program contains some procedures of which the body can be omitted while reading. The effect of these procedures is: the input procedure *readsymbol* gets, upon call, the value of the next symbol offered (in our case from a punched tape) in the code given above. The output procedure *write(number)* yields the value of *number* to an output mechanism (in our case a lineprinter). The input procedure *read* reads numbers and is used to fill the arrays *code* and *compression*. The procedure *EXIT* ends the program. Details about other undeclared procedures are given in the the manual of the MC-ALGOL system for the Electrologica X8 [3].

```
begin integer a, u, ea, q, qu;
  integer array code[0:127];
  begin integer k;
    for k:= 0 step 1 until 127 do code[k]:= read
  end;
  a:= 29; u:= 32; ea:= - 1; q:= 39; qu:= 60;
  begin integer letter, following letter, number of syllables;
    integer array compression[26:37,29:42];

    integer procedure readsymbol;
    begin integer k;
      k:= RESYM; if real letter(code[k]) then PRSYM(k);
      if k = 122 then EXIT; readsymbol:= code[k]
    end;

    procedure write(number); integer number;
    begin TAB; ABSFIXT(2, 0, number); NLCR end;

    Boolean procedure real letter(symbol); integer symbol;
    real letter:= symbol > 0 ∧ symbol < 63;

    Boolean procedure vowel(letter); integer letter;
    vowel:= letter < 39;

    Boolean procedure condensed(letter); integer letter;
    condensed:= letter < 26 ∨ (letter > 37 ∧ letter ≠ q);

    Boolean procedure condensing(letter); integer letter;
    condensing:= letter > 28 ∧ letter < 43;

    begin integer i, j;
      for i:= 26 step 1 until 37 do
      for j:= 29 step 1 until 42 do compression[i,j]:= read
    end;
    number of syllables:= 0; NLCR;
```

```
LETTER: letter:= readsymbol;
BEGIN: if real letter(letter) then
  begin if vowel(letter) ∨ letter = q then
    begin if condensed(letter) then
      begin number of syllables:= number of syllables + 1;
        goto LETTER
      end condensed vowel and thus syllable found
      else
      begin following letter:= readsymbol; if letter = q then
        begin letter:= if following letter = u then qu else
          following letter; goto BEGIN
        end treatment of q;
      COMPRESSION: if condensing(following letter) then
        begin letter:= compression[letter,following letter];
          if letter = ea then
          begin number of syllables:= number of syllables + 1;
            following letter:= readsymbol;
            if following letter = u then goto LETTER else
            begin letter:= a; goto COMPRESSION
            end ea was not followed by u
          end treatment of ea;
          if real letter(letter) then goto BEGIN
        end of compression;
        number of syllables:= number of syllables + 1;
        letter:= following letter; goto BEGIN
      end treatment of uncondensed vowel
    end treatment of vowel
  end treatment of real letter
  else
  begin if number of syllables ≠ 0 then
    begin write(number of syllables); number of syllables:= 0
    end output of syllable count
  end treatment of word delimiter;
  goto LETTER
end syllable count
end
```

TABLE 1
word token lengths

number of syllables: / word token length	0	1	2	3	4	5	6	7	8	9	total
1	.11	.14									.24
2	.47	18.94	.03								19.45
3	.15	21.65	.04								21.84
4	.02	9.25	1.93	.01							11.20
5		3.24	4.20	.08							7.52
6		.50	8.22	.64	.01						9.36
7		.12	4.73	1.71	.15						6.71
8			1.70	3.73	.31						5.75
9			.74	3.37	.98	.02					5.11
10			.24	1.99	1.35	.11					3.69
11			.07	.95	1.74	.33	.01				3.10
12			.05	.39	1.15	.37	.02				1.98
13			.00	.14	.68	.32	.05				1.19
14				.04	.37	.39	.07	.06			.94
15				.01	.14	.27	.14	.03			.59
16				.00	.06	.16	.16	.04			.42
17					.02	.07	.11	.03	.01		.23
18					.01	.06	.15	.04	.01	.01	.28
19						.02	.08	.05	.01	.00	.17
20						.00	.05	.05	.01	.00	.12
21						.02	.02	.05	.01		.10
total	.75	53.83	21.96	13.05	6.97	2.16	.86	.36	.05	.01	

TABLE 2
word type lengths

number of syllables:	0	1	2	3	4	5	6	7	8	9	total
word type length											
1	.06	.06									.12
2	.48	1.69	.04								2.20
3	.23	4.76	.06								5.06
4	.04	7.08	1.59	.01							8.72
5		4.13	5.79	.12							10.04
6		.91	11.24	.97	.02						13.14
7		.18	8.45	3.31	.23						12.17
8			3.56	7.28	.53						11.37
9			1.46	6.69	1.88	.06					10.09
10			.48	4.34	2.83	.20					7.85
11			.12	2.14	3.63	.39	.02				6.31
12			.06	.78	2.36	.82	.04				4.06
13			.01	.30	1.46	.80	.12				2.68
14				.08	.77	.70	.15	.09			1.78
15				.02	.33	.61	.30	.04			1.30
16				.01	.14	.38	.34	.10			.96
17					.04	.18	.27	.07	.02		.58
18					.03	.14	.32	.11	.02	.02	.65
19						.06	.20	.12	.02	.01	.40
20						.01	.11	.13	.03	.01	.29
21						.03	.05	.12	.03		.23
total	.82	18.80	32.86	26.04	14.25	4.37	1.92	.78	.13	.03	

1.1.2.3. Results

The mistakes found can be divided into four groups:

1. Foreign words (often occurring in Dutch text) as pe-a-ce. These lie outside our problem.
2. Words with a diaeresis, which was not punched in our material. This made België to have two syllables instead of three. A diaeresis will be punched as a nonmoving character in front of the vowel, or form a new character together with the vowel. In both cases a correct count is easily programmed. Our program counts correctly if the diaeresis is punched as an underlining. (By considering the underlining as a consonant, the hyphenation in section 1.1.3 will be correct too.)
3. Abnormal cases, like the proper name Wttewaal, which has three syllables. Note: abbreviations without vowels, like KLM, are considered as zero-syllabic words.
4. Real mistakes by wrong compression, like in the word museum which is counted as a 2-syllabic word instead of 3-syllabic.

The rarity of words like Wttewaal and museum permits us to say that we have found a satisfactory solution for the problem of automatic syllable counting in Dutch words.

1.1.2.4. Applications

1.1.2.4.1. Language statistics

A slightly changed program was used to process the newspaper words from a word count [4]. For each of ten newspapers the number of words occurring more than once, with a certain word length in oneletters (only the digraph ij was considered as one letter) and a certain number of syllables, was determined. This was done for the word tokens (each word as often as it occurs) and for the word types (each different word counted once). The percentages can be found in tables 1 and 2 on page 14 and 15.

The average word token contains 1.85 syllables and the average syllable from a word token contains 2.92 letters. The average word type contains 2.62 syllables and the average syllable from a word type contains 3.00 letters. The Dutch language therefore occupies in the list of Fucks [5] a place between German and Arabic. According to Fucks the frequency of an n-syllabic word would be equal to $c'c^{n-1}/(n-1)!$. A simpler formula is that this frequency equals c^{-n}(where c is about 2.05). In table 3 these two theoretical values are given next to the actual values found in our material. Monosyllabic words of 7 letters in our material were spreekt and slechts (schraalst is longer); the shortest word with nine syllables was antirevolutionaire.

The dent in the word length histogram at word length 5 is explainable by the transition from 1-syllabic to 2-syllabic words. As a measure for word length, the syllable count is more essential than the letter count (it is, e.g., invariant against spelling changes).

TABLE 3

percentage of n-syllabic words according to

Fucks	c^{-n}	observation	n
42.74	51.30	53.83	1
36.33	24.98	21.96	2
15.44	12.17	13.05	3
4.38	5.92	6.97	4
0.93	2.89	2.16	5
0.16	1.41	0.86	6
0.02	0.68	0.36	7
0.00	0.33	0.05	8
0.00	0.16	0.01	9

1.1.2.4.2. Poetical analysis

Although the spelling syllable cannot be equated to the poetical syllable, a slightly changed program was used to determine the number of syllables in lines from a poem. The same program also determines the rhyming schedule: capital letters refer to masculine rhyme, small letters to feminine rhyme (this last rhyme being defined by having an e in the last syllable).
An example:

A.C.W. Staring, HET HONDENGEVECHT	number of syllables	rhyme
Bereisde Roel zag op zijn tochten	9	a
geweldig veel. Twee bullebijters vochten,	11	a
voor 't wijnhuis, in een kleine Poolse stad,	10	B
terwijl hij juist aan 't venster zat:	8	B
"zulk vechten, mensen. -- Zij verslonden	9	c
malkander letterlijk. Met iedren hap, ging oor	12	D
of poot er af - en glad als vet er door.	10	D
Ons scheiden kwam te laat. Wij vonden	9	c
het restjen: - op mijn eer,	6	E
de staarten, en niets meer."	6	E

1.1.2.5. Other languages

The same procedure is possible for many other languages, with the exception of French where the compression cannot be done in the same way.

1.1.3. Division into syllables

1.1.3.1. Analysis

Given: a Dutch word in machine-readable form. Required: to place hyphens between the spelling syllables of the word.

Our solution arrived at after numerous improvements, resulting from the error analysis of the application to a standard list of words is:

1. Translate the oneletters into a suitable coding (defined in 1.1.2.1).
2. Condense the oneletters into generalized letters (as described in 1.1.2.1).
3. Cut off suffixes from the end of a word. The suffix heid, e.g., will make the division woest-heid instead of woes-theid. This is repeated until no suffix at the end of the word is found.
4. Look for the first pair of vowels. If there are no two vowels, then the process is completed. If they exist, then a hyphen is to be placed somewhere between them according to four criteria:
   - 4.1. Cut off useful prefixes: this will e.g., give on-eens instead of o-neens (do not always cut off on, otherwise: on-tzettend. Start cutting off the longest possible prefix).
   - 4.2. Certain vowels, e.g., aa, cannot, in general, occur as the last letter of a syllable. This rule gives raam-antenne instead of raa-mantenne. Certain other vowels, e.g., oei, are, in general, preferred as the last letter of a syllable. This rule gives vloei-stof instead of vloeis-tof.
   - 4.3. Between the two vowels stand 0,1, 2 or more consonants. We shall indicate a vowel by V, a consonant by C and a letter in general by X.
     - 4.3.1. No consonants between vowels: split V-V: papoe-a.
     - 4.3.2. One consonant between vowels: split V-CV: voe-ten.
     - 4.3.3. Two consonants between vowels: if the combination is contained in the list C2 of pronounceable pairs of consonants, then split V-CCV, else VC-CV. Thus: pro-bleem, but voer-den.
     - 4.3.4. More than two consonants between vowels. If the sequence

of the last three consonants is contained in the list C3 of pronounceable trigrams, then split -CCCV. Example: angst-schreeuw. Otherwise, if the last two consonants are in C2, then split -CCV. Example: angst-kreet. Otherwise split -CV, example: angst-gil.

4.4. Rules for special cases.

5. If in this way the leftmost syllable is found rule 4 is applied to the remaining part of the word.

The necessary information is therefore:

1. Knowledge of letters, vowels, syllable delimiters, etc.
2. The compression matrix.
3. The collection C2 of pronounceable CC-combinations.
4. The collection C3 of pronounceable CCC-combinations.
5. The suffixes.
6. The prefixes with the cases in which they have to be applied.

The first two groups have been discussed in 1.1.2.1. The last four groups are strongly dependent on each other. Each change in one has consequences for the other.

The collection C2 contains, on the basis of our newspaper material,

pl pr tr chr

bl br cl cr dr fl gr kl kr kw sc sch th vl wr zw

fr st sp.

The combinations

sf sj sk sl sm sn kn kj pn pj ts dw tw wr, and others (see 1.2.2), were not taken although they sometimes do occur at the beginning of a syllable. They would cause more incorrect splittings than correct ones. The last three elements of C2 (fr, st and sp) are only treated as elements of C2 when they are preceded by another consonant. If they occur between two vowels, then they are split into f-r, s-t, and s-p.

The collection C3 contains spl, spr, str and schr. These combinations consist of an s followed by one of the first four elements of C2.

The roles of the suffixes and prefixes are not symmetrical. This is caused by the fact that the left consonant part of a Dutch syllable is more predictable than the right consonant part (see 1.2) and because the useful suffixes do indeed occur at the end of a word, whereas prefixes may also occur in the middle of a word. Therefore, suffixes are cut off in the beginning of the process only, but prefixes are considered at every splitting. Certainly not all the conventional prefixes and suffixes have a useful effect. We confined ourselves to prefixes and suffixes of one syllable and of maximally four generalized letters (e.g., schier is allowed, but was not used, as a prefix).

The useful suffixes can be divided into three groups:

1. Those of which the first letter occurs as a last letter of an element from C2 or C3 (one could imagine suffixes of which the first two consonants appeared at the end of elements of C3, but this kind was not necessary).
   Example: the suffixes heid and wijs prevent the wrong splittings woes-theid and jaz-zwijs.
2. Suffixes starting with a CC-combination which is not element of C2, but which is pronounceable. Example: sfeer.
3. Suffixes starting with a vowel (they are dangerous to use).
   Example: aard prevents the splitting la-faard.

The useful prefixes can be divided into four types: VC, CVC, VCC and XXCC. Their usefulness depends on the position of the second vowel in the word region under consideration. If this vowel occupies the 3rd, 4th, 5th or 6th place, then the following splittings would be prevented by the following types:

	second vowel in position			
prefix type	3	4	5	6
VC	o-necht	a-flikken		
CVC		wa-norde	di-sponibel	di-splezier
VCC		al-sof	on-thouden	
XXCC			hoof-donderwijzer	aart-spiekeraar

Other considerations, however, are also involved in the decision whether a certain prefix is actually used.
A case in itself is presented by those prefixes that always have to be cut off, like hof (preventing ho-fauto). Also, one can introduce prefixes that are not used if immediately followed by the letter e. Then the prefixes rand and film would prevent ran-dapparatuur and fil-macteur but still deliver ran-den and fil-men. These possibilities were not used.

Dependent on the position of the second vowel, a certain part of the prefix list is searched. In this way the (on, ont) and the (voor, voort) problem is solvable in many instances.
We made use of the following suffixes:

aard heid laan lijk loos wijs ren rijk land ling ring

and the following prefixes:

aan oor in on oer er af ab
daar der hier voor ven ver her wan maat sub zuid dis
van nog hof jet
als ont
aarts voort hoofd noord groot

The changes we made in our successive attempts consisted mainly of omitting prefixes and suffixes that caused too many errors. For other languages, such as German,it was necessary to treat almost each prefix on its own, to consider 2-syllabic prefixes, and so on.
The choice of prefixes and suffixes is determined largely by the choice of C2 and C3. The strange prefixes der, ven, jet and ver were chosen to cut off the 2-syllabic prefixes on-der, bo-ven, sow-jet and o-ver. Mistakes caused by the occurrence of der as a prefix in words like studeren were corrected by taking ren as a suffix (suffixes have priority over prefixes).
Changes in these lists can be made without need to change the program. The information in C2, C3, prefixes and suffixes, is coded by treating a sequence of letters as a number in the 64-based number system.

Example: schr will be given the value: code(s) * 64 * 64 + code(ch) * 64 + code(r) = 221044.
Finally, some special measures are incorporated in the program. The method described so far would still make errors like: gek-neveld, bed-wongen, (ge and be cannot be taken as prefixes, but the list C2 is extended after them), bure-au,(the compression of e and au still has to be done), e-xamen, (there is no hyphenation at an x between vowels), meen-emen, (the rule that ee cannot be the end of a syllable has exceptions), indu-strie, (str between vowels is not always to be considered as an element of C3), per-sconferentie, (when sc follows a consonant the hyphenation will be between s and c), nation-aal, (the prefix on will not be applied if it follows the letter i), regering-sonderhandelingen, (The s following ng, followed by a vowel, will be part of the preceding syllable), toes-tanden, (the rule that st between vowels is split has exceptions).
By special measures the words in these categories are correctly hyphenated. Of general importance is the concept of suffixes without a definite ending and of prefixes without a definite beginning: stand was not considered as a suffix, but there always will be hyphenation before its occurrence; in the same way, hyphenation will always occur after ngs followed by a vowel.
No ad hoc measures were taken for one word only in our standard list. At one point the number of hyphenated parts differs from the number of syllables in 1.1.2: according to [1] there will be no hyphenation at an x between two vowels; examen has three syllables but is hyphenated as exa-men.
Hyphenation in which letters are deleted, as in the diminutive forms strootje, (stro-tje), menuutje (menu-tje) and opaatje (opa-tje) were not realized.

1.1.3.2. ALGOL program

As in the syllable counting case, the program is independent of the

installation used. The array *code* takes care of the coding of the letters. The array *WORD* contains the letters of the word to be hyphenated in the installation-dependent code, which can remain unspecified as these letters are put out in that same code, with hyphens at appropriate places. The letters, translated in our code and later condensed into generalized letters, are stored in the array *word*. Between *WORD* and *word*, a reference list is maintained. The input procedure *readsymbol* is given the value of the next character offered in the installation-dependent code. The output procedure *out(sign)* gives the character with installation-dependent code *sign* to an output mechanism. The hyphen is the only symbol whose installation-dependent code value we need to know. Luckily the hyphen has the same installation-dependent code value - 64 - as in our special code. For details on undeclared procedures we refer again to the manual [3]. For further details we refer to the ALGOL program SYLSPLIT on the next six pages.

```
begin integer a, d, e, k, n, o, p, s, t, u, x, be, ge, sc, st, gs,
  au, eau, mee, wee, zee, consonant, hyphen, end of text, no letter
  but word element;
  integer array code[0:127];

  integer procedure readsymbol;
  begin integer a;
  IN: a:= REHEP; if a = 0 ∨ a = 127 then goto IN; readsymbol:= a
  end;

  procedure out(sign); integer sign; PUHEP(sign);

  for k:= 0 step 1 until 127 do code[k]:= read; a:= 29; d:= 45;
  e:= 30; k:= 43; n:= 50; o:= 33; p:= 51; s:= 53; t:= 54; u:= 32;
  x:= 57; be:= 2846; ge:= 3038; sc:= 3432; st:= 3446; gs:= 3061;
  au:= 10; eau:= 23; mee:= 3163; wee:= 3611; zee:= 3739;
  consonant:= 59; end of text:= 66; no letter but word element:= 63;
  hyphen:= 64;
  begin integer first vowel, second vowel, word begin, word end, WORD
    begin, WORD end, symbol, letter, next letter, front, back, h, i, j;
    integer array word, WORD, reference from word to WORD[1:50],
    compression[26:40,29:42], prefixes[2:4,0:18],
    suffixes[3:4,1:8], C2[1:23];

    boolean procedure real letter(sign); integer sign;
    real letter:= sign < 63;

    boolean procedure vowel(letter); integer letter;
    vowel:= letter < 39;

    boolean procedure uncoondensed(letter); integer letter;
    uncondensed:= letter > 25 ∧ letter < 41;

    boolean procedure condensing(letter); integer letter;
    condensing:= letter > 28 ∧ letter < 43;

    boolean procedure aa group(letter); integer letter;
    aa group:= letter < 29 ∧ letter > 24;

    boolean procedure aai group(letter); integer letter;
    aai group:= letter < 6;
```

```
procedure search vowels;
begin boolean first vowel found;
  first vowel found:= false;
  for i:= word begin step 1 until word end do
  begin if vowel(word[i]) then
    begin if first vowel found then
      begin second vowel:= i; goto OUT end
      else
      begin first vowel:= i; first vowel found:= true end
    end
  end;
  for i:= WORD begin step 1 until WORD end do out(WORD[i]);
  out(symbol); i:= j:= 1; goto read word;
OUT:
end search vowels;

procedure cut off(h); integer h;
begin word begin:= word begin + h;
  for i:= WORD begin step 1 until reference from word to WORD[word
  begin] - 1 do out(WORD[i]); out(hyphen);
  WORD begin:= reference from word to WORD[word begin]; goto REST
end cut off;

procedure prefix(h); value h; integer h;
begin front:= word[word begin] × 64 + word[word begin + 1];
  if front = be ∨ front = ge then
  begin letter:= word[word begin + 2];
    if word end > 5 ∧ (letter = d ∨ letter = k ∨ letter = s ∨
    letter = t) then cut off(2) else goto OUT
  end;
  if h > 4 ∨ h < 2 then goto OUT;
  if h > 2 then front:= front × 64 + word[word begin + 2];
  if h > 3 then front:= front × 64 + word[word begin + 3];
  for j:= prefixes[h,0] step 1 until 18 do
  begin if front = prefixes[h,j] then cut off(h) end;
OUT:
end prefix;
```

```
  for i:= 26 step 1 until 40 do
  for j:= 29 step 1 until 42 do compression[i,j]:= read;
  for i:= 2, 3, 4 do
  for j:= 0 step 1 until 18 do prefixes[i,j]:= read;
  for i:= 3, 4 do
  for j:= 1 step 1 until 8 do suffixes[i,j]:= read;
  for i:= 1 step 1 until 23 do C2[i]:= read; i:= j:= 1;
read word: symbol:= readsymbol; letter:= code[symbol];
  if letter = end of text then EXIT; if real letter(letter) then
  begin WORD[i]:= symbol; word[j]:= letter;
    reference from word to WORD[j]:= i; i:= i + 1; j:= j + 1;
    if i = 51 then
    begin symbol:= readsymbol; goto too long
    end word too long;
  goto read word
  end real letter put in word and WORD ;
  if i = 1 then
  begin out(symbol); goto read word end;
  if letter = no letter but word element then
  begin WORD[i]:= symbol; i:= i + 1; goto read word end;
  if i = 2 then
  begin out(WORD[1]); out(symbol); i:= j:= 1; goto read word end;
too long: WORD end:= i - 1; word end:= j - 1;
  WORD begin:= word begin:= i:= j:= 1; h:= 0; goto word is read;
COMPRESSION: if uncondensed(letter) then
  begin next letter:= word[i + 1]; if condensing(next letter) then
    begin letter:= compression[letter,next letter];
      if letter > 0 then
      begin i:= i + 1; h:= h + 1; word[i]:= letter;
        goto word is read
      end
    end
  end compression;
  word[j]:= word[i]; j:= j + 1; i:= i + 1;
```

```
    reference from word to WORD[j]:= reference from word to WORD[j] + h;
  word is read: if i ≠ word end then
    begin letter:= word[i]; goto COMPRESSION end
    else
    begin word[j]:= word[word end]; word end:= j
    end word compression;
  compression done: search vowels; h:= 3;
    back:= word[word end] + 64 × word[word end - 1] + 4096 ×
    word[word end - 2];
  suffix: for i:= 1 step 1 until 8 do
    begin if back = suffixes[h,i] then
      begin word end:= word end - h; WORD end:= WORD end + 1;
        for j:= WORD end step - 1 until reference from word to
        WORD[word end + 1] + 1 do WORD[j]:= WORD[j - 1];
        WORD[reference from word to WORD[word end + 1]]:= hyphen;
        goto compression done
      end hyphen inserted
    end;
    if h = 3 then
    begin h:= 4; back:= back + 262144 × word[word end - 3];
      goto suffix
    end suffixes of four letters;
  REST: search vowels;
    if aai group(word[first vowel]) then cut off(first vowel - word
    begin + 1); if second vowel - first vowel = 1 then goto vowels;
    if second vowel - first vowel = 2 then goto one consonant;
    front:= word[second vowel - 2] × 64 + word[second vowel - 1];
    for i:= 1 step 1 until 23 do
    begin if front = C2[i] then
      begin if i < 5 ∧ word[second vowel - 3] = s then
        begin if word[second vowel - 4] = u then cut off(second vowel
          - word begin - 2); if word end > 16 ∧ i = 2 then
          begin if second vowel < word end - 2 ∧ second vowel > 9
            then cut off(second vowel - word begin - 2)
          end;
```

```
        prefix(second vowel - word begin - 2);
        if aa group(word[first vowel]) ∧ second vowel - first
        vowel = 4 then cut off(firstvowel - word begin + 2) else
        cut off(second vowel - word begin - 3)
      end;
      if aa group(word[first vowel]) ∧ second vowel - first vowel
      = 3 then cut off(first vowel - word begin + 2);
      prefix(second vowel - word begin - 1); if i > 20 then
      begin if first vowel = second vowel - 3 ∧ ¬(front = st ∧
        word[second vowel] = a ∧ word[second vowel + 1] = n ∧
        word[second vowel + 2] = d) then cut off(second vowel -
        word begin - 1)
      end;
      if front = sc then
      begin if ¬vowel(word[second vowel - 3]) then cut off(second
        vowel - word begin - 1)
      end SC;
      if second vowel - first vowel = 3 ∧ word[first vowel] = o ∧
      word[first vowel + 1] = p then cut off(2);
      cut off(second vowel - word begin - 2)
    end
  end;
  if front = gs then
  begin if word[second vowel - 3] = n then cut off(second vowel -
    word begin)
  end NGS-vowel;
  prefix(second vowel - word begin - 1);
  prefix(second vowel - word begin - 2);
  prefix(second vowel - word begin);
  cut off(second vowel - word begin - 1);
vowels: if word[first vowel] = e ∧ word[second vowel] = au then
  begin word[first vowel]:= eau; word end:= word end - 1;
    for i:= first vowel + 1 step 1 until word end do
    begin word[i]:= word[i + 1];
      reference from word to WORD[i]:= reference from word to WORD[i + 1]
```

```
        end;
        goto REST
      end EAU-compression;
      cut off(first vowel - word begin + 1);
    one consonant: if word[first vowel + 1] = x then
      begin word[first vowel]:= consonant; goto REST end;
      if first vowel > word begin ∧ aa group(word[first vowel]) then
      begin front:= 64 × word[first vowel - 1] + word[first vowel];
        if front = mee ∨ front = wee ∨ front = zee then cut off(first
        vowel - word begin + 1) else cut off(first vowel - word begin + 2)
      end EE;
      if first vowel > 1 then begin if vowel(word[first vowel - 1]) ∧ word[word
      begin] = o ∧ word[word begin + 1] = n then cut off(1) end;
      if second vowel < word end then prefix(second vowel - word
      begin); cut off(second vowel - word begin - 1)
  end SYLSPLIT
end
```

1.1.3.3. Results

The program SYLSPLIT was applied to the 43 712 word tokens (4 114 types) of [4]. Several methods to measure the error rate in hyphenation are possible [6]. One can consider the percentages of incorrectly hyphenated words, or of incorrect hyphen positions. This may be done for word tokens, or for word types. In our test material the number of incorrectly split words was equal to the number of incorrect hyphen positions, no word had more than one mistake. We found these numbers of errors:
64 of the 4 114 word types were hyphenated incorrectly: 1.6%
224 of the 43 712 word tokens were hyphenated incorrectly: 0.5%
224 of the 28 229 hyphens were positioned incorrectly: 0.8%
(This error measuring method evidently takes for granted that the correct number of syllables of each word has been found. In other languages this situation may not exist).
There are three ways to categorize the 64 errors: according to the number of consonants between vowels, whether the words are compounds, and whether a critical s is present.

type	number	example
1. according to the number of consonants between vowels:		
VCV	22	doe-leinden
VCCV	16	ko-plamp
V...CCCV	26	beroep-strots
2. whether the words are compounds:		
compounds whose second member begins with a vowel:	22	slach-toffer
compounds with a critical s:	15	koer-speil
other compounds	12	werel-drecord
total compounds	49	
no compounds	15	

(of which		
by incorrect application of prefixes	11	Af-rika
by incorrect application of suffixes	2	verkl-aard
by old spelling forms with ee	2	Heer-enveen)

3. The presence of a critical s:
   In 21 cases the s made the error.
   In 43 cases no s was involved.

From these results it appears that the most dangerous words to hyphenate are compounds (without a dictionary a compound can only be predicted because of its greater length) and the words with an s. If one prefers to have no hyphenation above incorrect hyphenation, then one could refuse to hyphenate in the presence of an s. This strategy of not splitting in certain cases can be improved.
Compounds constitute the greatest source of errors in automatic hyphenation, and they will continue to do so however refined the program is made. The Reifler-procedure [13] to decompose compounds into components with the aid of a list of all components has been found to be effective for Dutch. This method consists, in short, of cutting off the longest fitting leftmost part of the compound which can be found in the list of components, and proceeding thus until no fitting part can be found; then the last decision is revoked and the last found component is replaced by a shorter one, until the whole compound is divided into parts which are elements of the component list.
A full solution of the hyphenation problem can thus only be given if a list of all component words in Dutch is given as information to the program. The minimal length of such a list remains an open problem.
The error rate of our automatic hyphenation program is low enough to make practical applications possible.

An example of a text with especially hard words hyphenated by SYLSPLIT:

Kwarts-la-gen ma-ken kwarts-la-gen. De twee-de zee-man veeg-de met een mee-ge-no-men zeem-lap leem-aar-de van het tweed-kleed-op-per-vlak. Pre-mier De Quay geeft freu-le Wtte-waal eau-de-co-log-ne en een skij-um-per. de he-ren Van Eijs-den, Krae-mer, Baay-en en De Bruyn her-in-ne-ren zich de hors d'oeu-vre. De bes-te ges-te bij ge-ste-gen be-ste-din-gen is een be-zui-ni-gings-actie. De woes-te toe-stand van de vloei-stof on-der de mi-cro-scoop was een on-ont-koom-baar suc-ces op de in-ter-na-ti-o-na-le pers-con-fe-ren-tie.

1.1.3.4. Applications

1.1.3.4.1. The most important technical application of automatic hyphenation lies in the automatic typesetting of texts. If copy is first punched without caring about line length, indentation, ligatures, etcetera, then a computer program can produce a punched tape to feed a typesetter. The only real problem for texts is the hyphenation of some words at the end of a line.

Many newspapers in the Netherlands now make use of the program described above [7]. The hyphenation problem in this application can be termed differently than we did before: it is no longer necessary to know all hyphen positions in a certain word, but it is sufficient to know the most reliable hyphen position in a certain portion of the word (the length of this portion will in general vary with the length of the typesetting line). We already mentioned the evasion around the letter s. The role of prefixes and suffixes can be changed from instructions to cut them off to warning instructions: hyphenation in this neighbourhood is dangerous.
The error percentages in 1.1.3.3 can be lowered by applying this strategy, but on the other hand the words which need to be hyphenated at the end of a line will tend to be the longer ones, which makes the error percentages for this application rise again.
Experimentally, a booklet [8] with 320 hyphenated lines was found to contain 3 errors. The rate of about 1% errors was also found by the newspapers using different (usually machinecoded) versions of SYLSPLIT.

1.1.3.4.2. A computer-made inventory of the Dutch syllables with their frequencies and an investigation into the way these syllables are constructed (see 1.2) is now made possible.

1.1.3.5. Other languages

After the program for Dutch we made similar programs for other languages. German [9] and Swedish [10] appeared to present the same problems and results comparable to Dutch. For the Romance languages [11], a solution which essentially only required the knowledge of C2 yields acceptable results. American English [12] is a case on its own; the best program obtained until now still has a 14% error rate. We give on page 35 short examples of the automatic hyphenation in each of the eight languages.

English

Ar-ticle 1. All hu-man be-ings a-re born free and equal in dig-ni-ty and rights. They are en-dowed with reas-on and con-science and should act to-wards one anoth-er in a spir-it of broth-er-hood.

Swedish

Ar-ti-kel 2. En-var är be-rät-ti-gad till al-la de fri- och rät-tig-he-ter, som ut-ta-las i den-na för-kla-ring, u-tan åt-skill-nad av nå-got slag, så-som ras, hud-färg, kön, språk, re-li-gi-on, po-li-tisk el-ler an-nan upp-fatt-ning, na-ti-o-nellt el-ler so-ci-alt ur-sprung, e-gen-dom, börd

German

Ar-ti-kel 4. Nie-mand darf in Skla-ver-ei o-der Lei-bei-gen-schaft ge-hal-ten wer-den; Skla-ver-ei und Skla-ven-han-del sind in al-len ih-ren For-men ver-bo-ten.

Ar-ti-kel 5. Nie-mand darf der Fol-ter o-der grau-sa-mer, un-mensch-li-cher o-der er-nie-dri-gen-der Be-hand-lung o-der Stra-fe un-ter-wor-fen wer-den.

Italian

Ar-ti-co-lo 7. Tut-ti so-no e-gua-li di-nan-zi al-la leg-ge e han-no di-rit-to, sen-za al-cu-na dis-cri-mi-na-zio-ne, ad u-na e-gua-le tu-te-la da par-te del-la leg-ge. Tut-ti han-no di-rit-to ad u-na e-gua-le tu-te-la con-tro

French

Ar-ti-cle 8. Tou-te per-son-ne a droit à un re-cours ef-fec-tif de-vant les ju-ri-dic-tions na-tio-na-les com-pé-ten-tes con-tre les ac-tes vio-lant les droits fon-da-men-taux qui lui sont re-con-nus par la cons-ti-tu-tion ou

Spanish

Ar-ti-cu-lo 9. Na-die po-drá ser ar-bi-tra-ria-men-te de-te-ni-do, pre-so ni des-te-rra-do.

Ar-ti-cu-lo 10. To-da per-so-na tie-ne de-re-cho, en con-di-cio-nes de ple-na i-gual-dad, a ser oí-da pú-bli-ca-men-te y con jus-ti-cia por un tri-bu-nal

Rumanian

Ar-ti-co-lul 11. O-ri-ce per-soa-nă a-cu-za-tă de co-mi-te-rea u-nei in-frac-ţiuni es-te pre-zu-ma-tă ne-vi-no-va-tă, cît timp vi-no-vă-ţia sa nu a fost do-ve-di-tă, po-tri-vit le-gii, printr-o ju-de-ca-tă pu-bli-ca la ca-re să i

Portuguese

Ar-ti-go 12. Nin-guém se-rá su-jei-to a in-ter-fe-rên-cias na sua vi-da pri-va-da na sua fa-mí-lia, no seu lar ou na sua cor-res-pon-dên-cia, nem a a-ta-ques à sua hon-ra e re-pu-ta-ção. To-do ho-mem tem di-rei-to

1.2. Quantitative analysis of the Dutch syllable

1.2.1. Introduction

One of the results of a Dutch word count [14], is a list of the 5 000 syllable types found among the 82 000 syllable tokens derived from the 44 000 word tokens in newspaper texts. The list of syllable types was further analyzed with the aid of a computer. In this investigation of the structure of the Dutch spelling syllable, three points of departure are of importance:

1. We use fixed and generally available basis material, viz. the 44 000 newspaper words from [14] as split into syllables by the program in 1.1.3. No call on the imagination was made to think up special words. The disadvantage of this point of departure is clear: the words chloor and erwt did not occur in our material and therefore the combinations chl and rwt will not occur as consonant combinations in the beginning or ending of a syllable. Our sample of 82 000 syllable tokens appeared, however, large enough to give results of general interest.
2. We treat the spelling syllable. The phonetic or phonological syllable structure can, however, to a large extent be derived from the spelling structure. The ordering of the results makes it possible, e.g., to consider ch and ng, as one element as has been the case with the digraph ij. Therefore, our results may also be of importance for the knowledge about phonetic or phonological syllables. If basis material of phonetic or phonological syllables is available in some discrete alphabet, then the same program for structure analysis as we employed can be used.
3. Our aim is a quantitative determination of the structure of the Dutch spelling syllable tokens. Not the question whether something occurs but how often it occurs is under discussion. The structure of Dutch syllable types, at least of the monosyllabic monomorfemic substantives, has been investigated by Oosterlynck [15].

This principle has no disadvantage: the structure of syllable types is directly derivable from that of the syllable tokens.
These three points of departure are closely linked with our tool: the computer, which can only operate in formal ways.
Concerning principle 1: the computer has no knowledge of Dutch and we are thus committed to our basis material which says by definition what is Dutch and what is not. To count by hand a sample of this size is not impossible but inhuman work, that can be done faster and more reliably by a computer.Concerning principle 2: the spelling syllable, that strange product of contradictory phonetic and phonological motives, plus traditional agreements, was determined by computer with 99% reliability. Using a phonetic alphabet, a program delivering the phonetic syllables would be more simple. Written languages has its own laws and forms a legitimate domain for research. Without any reference to phonetics, concepts like "vowel", "consonant" and "syllable" may be defined on purely graphemics grounds. The vowels are, e.g., a minimal subset of the alphabet such that each word from a vocabulary contains at least one element from that subset. If one applies this definition to our material, then the period (in abbreviations) and the apostrophe are vowels. The abbreviations were removed from the basic material; the frequency of the apostrophe was low enough to neglect it.
Concerning principle 3: the advantage of automatic methods is clear. Perhaps superfluously, we emphasize that our points of departure are not a consequence of the wish to find applications for a computer; on the contrary, the three principles are valid per se, but are easier adhered to by computers than by men.
The results were found formally, except on one point: the correction of the one percent faultily split syllables.
The results were grouped by informal methods, such as neglecting abnormal cases and ordering combinations in convenient tables.
Since our tool, the computer, will no longer be mentioned in the following sections, it may be useful to sketch the whole process:

1. The texts from 10 newspapers of one day in 1956 (44 000 words) were punched on paper tape.
2. The words were alphabetised and counted. There appeared to be 9 000 word types.
3. The word list was divided into syllables by SYLSPLIT.
4. The syllables were alphabetised and counted, account being taken of the frequency of the words from which they came. Four kinds of syllables (monosyllabic words, initial syllables, middle syllables, and end syllables) were differentiated: 5 000 types. If we do not make this division into four kinds, then 3 000 types appear to exist. These four steps are described in [14], which also contains the resulting list.
5. The syllables were split into: consonant clusters in the beginning (CL-group), vowels (V-group) and consonant clusters at the end (CR-group). Elements of CL and CR may contain the hyphen to show from what type of syllable they are derived.
6. The elements of CL, V and CR were alphabetised and counted, account being taken of the frequencies of the syllables from which they came.

In 1.2.2 the elements of CL, in 1.2.3 the elements of V and in 1.2.4 the elements of CR are described. All frequencies are relative frequencies in 10 000. If a type does occur, but less than 5 times in the 82 048, then it has the relative frequency 0. If a type does not occur at all, then this is indicated by "-".
The connection between the three groups CL, V and CR is broken in this analysis. Our material would enable us to study that connection (especially strong between V and CR), e.g., in the case of monosyllabic words.

1.2.2. Consonants in the beginning of a syllable

The 82 048 tokens of the CL-group gave 83 CL-types. One of these was the empty type (syllable starting with a vowel), 61 were normal types and 21 were considered abnormal types. Of the 83 types, 78 occurred without

preceding hyphen (from initial syllables, indicated by the + sign) and 65 with preceding hyphen (from non-initial syllables, indicated by the - sign). The CL-types found are given with their relative frequency in table 4 (the apostrophe was neglected 7 times).
Abnormality, and this goes for sections 1.2.3 and 1.2.4 too, may be assigned to types with a frequency below 5 in 10 000. Whether they are actually considered abnormal depends on the question whether they fit into a simple grammar. Sometimes it was considered from which words these abnormal combinations came: proper names, old spellings, foreign words. In CL the following 21 types were considered abnormal:
ck, cn, cqu, dj, gh, khs, lh, ph, rh, schm, sg, sh, sc, squ, sw, sz, thl, wh, wtt, zl, y
with a total frequency of 9 in 10 000. The combination zl occurred 13 times in our material, every occurrence from the Polish word zloty.
All other abnormal combinations had a frequency of 0 or 1 in 10 000.
The remaining 61 normal types can be ordered conveniently as in table 5 on page 41. According to the number of letters, CL can be divided into four groups: of 0, 1, 2 or 3 letters (counting the digraph ch as 1 letter). The number of types and tokens for these four groups can be found in the first two columns of table 11 on page 48.
CL appears to have a rather simple structure.

TABLE 4: CL-elements

type	+	−	total	type	+	−	total
empty	1230	183	1413	rh	0	0	0
b	221	100	320	s	44	161	205
bl	13	18	31	sc	1	0	1
br	17	20	37	sch	23	77	100
c	50	65	116	schm	0	0	0
ch	4	9	13	schr	5	5	10
chr	5	0	5	sf	0	−	0
ck	−	0	0	sg	0	−	0
cl	1	2	3	sh	1	−	1
cn	0	−	0	sj	2	−	2
cqu	−	0	0	sk	0	−	0
cr	1	3	4	sl	11	18	30
d	754	582	1336	sm	2	0	2
dj	1	−	1	sn	1	1	3
dr	13	22	35	sp	13	5	18
dw	1	2	2	spl	−	0	0
f	31	37	69	spr	5	9	15
fl	2	1	4	squ	0	−	0
fr	11	2	13	st	67	92	160
g	280	362	642	str	10	12	22
gh	0	0	1	sw	1	−	1
gl	2	0	2	sz	0	−	0
gr	31	18	49	t	211	503	713
h	416	100	517	th	9	5	14
j	42	32	74	thl	−	0	0
k	95	225	320	tj	0	2	2
khr	0	−	0	tr	14	36	50
kl	9	11	20	tsj	0	−	0
kn	1	0	1	tw	13	1	14
kr	22	3	25	v	485	219	705
kw	8	2	10	vl	8	4	12
l	100	433	533	vr	27	9	36
lh	0	−	0	w	274	102	376
m	262	208	470	wh	0	−	0
n	188	248	435	wr	1	0	1
p	73	94	167	wtt	0	−	0
ph	1	−	1	x	−	3	3
pl	14	7	21	y	0	−	0
pr	38	12	50	z	248	137	385
ps	1	0	1	zl	2	−	2
qu	1	4	5	zw	7	3	10
r	110	253	363				
				total	5536	4464	10000

TABLE 5: normal CL-elements

initial letter(s) followed by:	r	l	w	n	other	total
z 385			10			395
d 1336	35		2		tj 2	1375
t 713	50		14		th 14	791
					tsj 0	
k 320	25	20	10	1		376
c 116	4	3			(ch)	123
v 705	36	12				753
b 320	37	31				388
f 69	13	4				86
p 167	50	21			ps 1	239
sp 18	15	0				33
g 642	49	2				693
ch 13	5					18
sch 100	10					110
st 160	22					182
w 376	1					377
s 205		30		3	sj 2 sf 0 sk 0 sm 2 (sp sch st)	242
h 517						517
j 74						74
l 533						533
m 470						470
n 435						435
qu 5						5
r 363						363
x 3						3
total 8045	352	123	36	4	21	8581
				empty	type	1413

1.2.3. Vowels

The 82 048 tokens of the V-group (in which by definition of the syllable the empty one could not occur) yielded 37 types. Of these, 18 were rare (frequency of each below 10 in 10 000,total frequency 35), and one was considered abnormal (ye from the geographical name Yerseke).
The V-types found are given with their relative frequency in table 6 on page 43. As the diaeresis is not punched, the frequency of the ie is too high and those of i and e too low. The rank order of the vowels is, however, correct as given.
Five u's from qu and squ were already counted in CL.
According to the number of letters, V can be divided into three groups: of 1, 2 or 3 letters (counting the digraph ij as 1 letter). The number of types and tokens from these 3 groups can be found in the two middle columns of table 11 on page 48.

TABLE 6: V-elements

alphabetically		decreasing frequency	
a	1388	4089	e
aa	378	1388	a
aai	1	1021	i
aay	1	944	o
ae	1	409	ee
ai	10	378	aa
au	17	368	ij
ay	1	318	ie
e	4089	236	oo
eau	3	236	u
ee	409	152	oe
eeu	1	123	ei
ei	123	117	ui
eu	32	79	ou
eui	1	32	eu
ey	2	23	y
i	1021	20	uu
ie	318	17	au
ieu	15	15	ieu
ij	368	10	ai
o	944	6	oei
oe	152	4	ooi
oei	6	3	eau
oeu	0	2	ey
oey	0	2	oi
oi	2	1	aai
oo	236	1	uy
ooi	4	1	eeu
ou	79	1	aay
oui	0	1	ay
oy	1	1	eui
u	236	1	oy
ui	117	1	ae
uu	20	0	oeu
uy	1	0	oui
y	23	0	oey
ye	0	0	ye

1.2.4. Consonants at the ending of a syllable

The 82 048 tokens of the CR-group yielded 160 types. One of these was the empty type (syllable ending with a vowel), 110 were normal types, and 49 were considered abnormal types. Of the 160 types, 139 occurred without following hyphen (from ending syllables, indicated by a + sign) and 97 with following hyphen, indicated by a - sign. The great number of abnormal elements (according to types, not great according to tokens) can be ordered, as in table 7 on page 45. To order the 110 normal types, we neglect the apostrophe and consider ch, ng and dt as generalized letters.
Table 8 on page 45 gives the oneletter elements of CR with their relative frequencies.
Table 9 on the pages 46 and 47 gives the twoletter elements, ordered firstly according to second letter t, d or s, and secondly according to first letter l, r, m or n. This table also indicates which 2-letter combinations can still be followed by st, s, t or d. The frequencies of the 3-letter combinations thus obtained are given in table 10 on page 48. There are two types (and two tokens) of four letters: ndst and rnst, each of relative frequency zero.
According to the number of letters, CR can be divided into five groups: of 0, 1, 2, 3 or 4 letters (counting ch as one letter). The numbers of types and tokens for these five groups can be found in the last two columns of table 11 on page 48.
The group CR shows a much richer structure than CL: twice as many types and twice as many normal types.

TABLE 7
abnormal CR-elements

types	tokens		
3	6	caused by "ch":	rsch,ndsch,sch
5	8	caused by "c":	ck,nck,cks,ct,nc
11	6	caused by "h":	hn,hr,sh,th,h,lth,ngh,rth,ght,hms,sh
3	4	single letter:	j,v,z
7	6	double letter:	ff,kk,ll,nn,ss,tt,pp
4	2	second letter "z":	cz,dz,nz,tz
3	1	second letter "n":	jn,tn,wn
2	1	second letter "s":	js,vs
2	0	second letter "l":	ml,rl
3	0	two other letters:	ph,sj,sk
6	2	more than two letters:	wcz,rld,sjts,tts,rls,sts
49	37	total	

TABLE 8
one-letter CR-elements

type	+	-	total
n	1525	572	2097
r	424	535	959
t	494	106	600
s	168	157	325
l	129	177	306
k	132	62	194
g	100	50	151
m	67	77	143
ng	123	13	137
p	85	49	133
d	92	35	127
f	43	45	88
ch	28	45	73
w	18	13	31
c	2	21	23
b	3	17	21
dt	4	-	4
total	3437	1979	5416

TABLE 9
two-letter CR-elements

34 types with second letter "t","d" or "s",of which 12 also in table below

first letter:	l	r	m	n	t	d	s	b	f	p	g	k	w	ch	ng	total
t +							32	0	38	3	9	15	0	35	1	132
t −							10	−	0	−	−	−	−	28	−	39
t third							42	0	38	3	9	15	0	63 s	1	170
d +							1		1		13		2		1	18
d −							−		4		1		−		−	5
d third							1		5		13		2		1	22
s +					12	10			1	1	0	6	1		11	41
s −					9	7			2	2	2	1	1		5	29
s third					20 t	17			2 t	2	2 t	7 t	2		15 t	67
total				(496)	20	17	43	0	45	5	25	22	4	63	17	260

34 types with first letter "l","r","m" or "n",of which 12 also in table above

second:	m	n	t	d	s	b	f	p	g	k	w	ch	dt	x	total
l +	2		6	88	31		5	1	2	4			1		140
l −	1		1	5	3		3	0	1	1			−		15
l	3		7	93	34		9	2	3	6			1		155
third	s				t		t,s	t	d,t	s					
r +	2	1	22	47	48		1	2	5	11	−	−	13		152
r −	2	3	8	3	3		0	0	1	6	0	0	0		26
r	4	4	30	51	51		1	3	6	18	0	0	13		178
third	d	s,st	s	s	t		d	s	d,s	t,s		t			
m +			6	4	2	−	0	1							13
m −			0	0	0	0	−	1							2
m			6	4	2	0	0	2							15
third				st	t	t		t							
n			34	77	47					4		0	2	0	165
n −			22	30	9					7		−	−	−	68
n			56	107	56					11		0	2	0	233
third			s	s	t					s,t					
total	8	4	99	254	143	0	10	6	8	34	0	0	16	0	581

Total: 56 two-letter types, 841 two-letter tokens

TABLE 10
three-letter CR-elements
for structure see table 9

type	+	−	total
lfs	2	−	2
lks	1	1	2
lms	0	−	0
rps	−	0	0
rds	0	−	0
rts	2	1	3
rks	−	0	0
rgs	−	0	0
rns	0	−	0
nds	4	4	8
nts	0	0	1
nks	1	1	2
chts	2	5	6
total	12	12	25
13 types ending in "s"			
fst	0	0	0
tst	1	0	1
kst	1	0	1
gst	0	0	1
mst	2	0	2
nst	5	2	6
lst	0	0	0
rst	5	0	5
ngst	1	0	1
total	15	2	18
9 types ending in "st"			
lft	1	0	1
lpt	0	−	0
lgt	1	−	1
lgd	1	−	1
rfd	−	0	0
rkt	5	0	6
rgd	1	0	1
rmd	−	0	0
rcht	−	0	0
mbt	0	0	0
mpt	0	0	1
nkt	1	−	1
total	10	2	12
12 types ending in "d" or "t" (not "st")			

TABLE 11
types and tokens in CL, V and CR

	CL		V		CR	
	ty	to	ty	to	ty	to
0 letters	1	141	−	−	1	365
1 letter	20	776	7	807	16	528
2 letters	36	77	8	190	52	95
3 letters	5	5	11	3	39	9
4 letters	−	−	−	−	3	0
total	61	999	36	1000	111	996
abnormal	21	1	1	0	49	4
total	83	1000	37	1000	160	1000
(ch and ij considered as one letter.)						

1.3. Synthesis of the Dutch syllable

1.3.1. Generative grammar of the Dutch syllable

The letter sequences which might be Dutch words (e.g., new trade names) can be generated by a context-free grammar. The grammar in table 12 on page 51 is based on the results about CL, V and CR as given in 1.2. It would be best to generate the four kinds of syllables (monosyllabic words, initial syllables, middle syllables, and final syllables) separately. The grammar would then start with:

WORD → MONOSYL, INITIALSYL - MIDDLE - FINALSYL

MIDDLE → MIDSYL (-MIDDLE)

MONOSYL → ...

MIDSYL → ...

FINALSYL → ...

To each alternative in a right-hand side we could give a number, expressing the probability, based on the results in 1.2. The grammar's initial symbol is "WORD" and one applies the rewriting rules until only terminal elements (denoted by small letters and the hyphen) remain in the produced string. Nonterminal categories are denoted by capital letters.

The grammar is context-free (the language produced is even finite-state) but contains rules of type A → ∅ (zero-element) and there are letter sequences with more than one derivation.

From the grammar it appears that the number of different syllables is at most 430 000. In reality this number is, of course, much lower: in our material of 44 000 words we found 3 000 syllable types.

The grammar of table 12 has three claims:

1. Each Dutch word can be generated by the grammar.
2. Each CL, V and CR combination generated by the grammar occurs as such in a Dutch word.
3. The alternatives are ordered in decreasing probability. To each alternative we could give a number expressing the probability, based on the results in 1.2. These weighted rules then produce syllables in which the CL, V and CR have frequencies as found in Dutch texts.

A claim of each grammar is its simplicity, although no objective criterion for simplicity of a grammar is known. The linguistics relevance of a grammar is intimately connected to this simplicity.
Another important demand for a generative grammar is that each generated string is indeed acceptable in the language. This demand is not realistic for a grammar of the Dutch word, or even the Dutch syllable. A better grammar than the one in table 12 is, however, conceivable, especially one accounting for the restrictions between V and CR. This would diminish the number of produced syllables. The fewer letterstrings a word grammar, satisfying the three claims above, generates, the better it is.
We give an example of a shortened derivation:

WORD	rule
SYLLABLE	1
CL V CR	2
s POSTS aa Y I POSTL	3,10,13
s ch r aa I POSTMN	5,11,21,22
s ch r aa I T	24
schraaIst	26

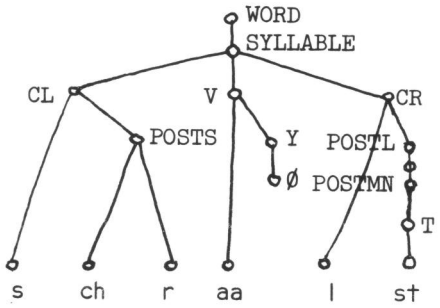

TABLE 12 : word grammar

1	WORD	→ SYLLABLE (-WORD)
2	SYLLABLE	→ (CL)V(CR)
3	CL	→ CL1,sPOSTS,ANTERr,ANTELl,ANTEWw,ANTENn,t(s)j
4	CL1	→ d,t,g,v,l,h,m,n,w,z,r,b,k,s,p,c,f,j,ch,th,x,qu,ps,y
5	POSTS	→ t(r),ch(r),p,l,n,j,f,k,m,c,qu,ph
6	ANTER	→ ANTEL,t,d,w
7	ANTEL	→ ANTEN,sp,b,v,ch
8	ANTEW	→ t,k,z,d
9	ANTEN	→ p,g,k,f
10	V	→ e(e(u)),aY,i(e(u)),oUI,aaY,ij,uY,oo(i),oeY,eUI,y
11	Y	→ (i),y,u
12	UI	→ Y,ui
13	CR	→ CR1,ANTEDd,ANTETt,rPOSTR,lPOSTL,ANTESs(t),nPOSTN,mPOSTM
14	CR1	→ n,r,t,s,l,k,g,m,p,d,f,(s)ch,w,c,b,x
15	ANTED	→ ANTEDS,s
16	ANTEDS	→ f,g,w,b
17	ANTET	→ ANTETS,s,(s)ch,x
18	ANTETS	→ ANTEDS,d,k,p
19	ANTES	→ ANTETS,t
20	POSTR	→ POSTL,nD,wD
21	POSTL	→ POSTN,POSTM,mD
22	POSTN	→ POSTMN,gD,kT,chT
23	POSTM	→ POSTMN,pT,fD
24	POSTMN	→ dT,s(d),T,tst
25	D	→ T,d
26	T	→ (s)(t)

1.3.2. Machine produced words

The grammar of 1.3.1 enables us to generate syllables. The notation of the rules of a context-free grammar and the procedure mechanism in ALGOL 60 make it possible to transcribe a program directly from the grammar. Each syntactic category becomes a procedure. The procedures belonging to the terminal elements have the effect that those terminal elements are printed on the lineprinter. For example, the grammar rule
ANTES → ANTETS, †
is transcribed into:

```
procedure ANTES; if RANDOM < 0.75 then ANTETS else t;
```

RANDOM is a real procedure delivering a random number between 0 and 1. The effect of a call of the procedure *ANTES* is that, with a probability 0.75, the procedure *ANTETS* is called or, with a probability 0.25, the letter † is given to an output mechanism.
The program "word synthesis" consists of a repeated call of the procedure *WORD*. The first grammar rule
WORD → SYLLABLE (-WORD)
is transcribed into:

```
procedure WORD; begin SYLLABLE; if RANDOM < 0.45 then WORD end;
```

For the probabilities found in 1.2 and for a certain sequence of random numbers, the program gave the first series of words in table 13 on page 53. Here, the four kinds of syllables are still treated in the same fashion. By changing the probabilities for initial and non-initial CL-types and for final and non-final CR-types, the last series of words in table 13 was generated.

TABLE 13

Words generated without differentiating four kinds of syllables:

zwer
haan-tek
vest
daf
wa-gop-hir-wir
er-da-daap
gaa
ij-lun
non-ge
ra-wa
loen
ben-des-ver-zi
o-ke-aan
ver-mon
ijn-ijn
gin
men-si
en
no
ram

Words generated taking account of the four kinds of syllables:

jengd
win
y-er-lan
lad-peus-de
go-ses-lin
in
geer-veem-dib-nen
rer
viet
riech
von
ee-vat
ve
dir
dir
bon
an-lird-feir-res-laam-en
og
vijr
ne-en-jil-maain

Sources Chapter 1

[1] Woordenlijst van de Nederlandse Taal, Nijhoff, 's-Gravenhage 1954.

[2] P. Naur (ed), Revised Report on the Algorithmic Language ALGOL 60, Regnecentralen, Copenhagen 1962.

[3] F.E.J. Kruseman Aretz, Het MC-ALGOL 60-systeem voor de X8, MR 81, Mathematical Centre, Amsterdam 1966.

[4] J.A.Th.M. van Berckel, Onderzoek Woordfrequentie, R 642/2, Mathematical Centre, Amsterdam 1962.

[5] W. Fucks, Theorie der Wortbildung, Math. Phys. Semesterberichte IV, 1954.

[6] L. Molyneux, An index of the quality of a hyphenation algorithm, internal report Computer Typesetting Research Project, Newcastle 1969.

[7] Automatisering in de grafische industrie, Techniek en Toepassing, Philips Eindhoven, nr. 25, 1967.

[8] First Lustrum publication of the Dutch Computer Society N.R.M.G., Amsterdam 1964.

[9] H. Brandt Corstius and E.G.M. Broerse, Automatisch scheiden van Duitse lettergrepen, NR 2 Mathematical Centre, Amsterdam 1967.

[10] - , Automatisch scheiden van Zweedse lettersgrepen, NR 5 Mathematical Centre, Amsterdam 1968.

[11] - , Automatisch scheiden van lettergrepen in Romaanse talen, NR 4 Mathematical Centre, Amsterdam 1968.

[12] - , Automatisch scheiden van Amerikaans-Engelse lettergrepen, NR 5 Mathematical Centre, Amsterdam 1968.

[13] E. Reifler, Mechanical Determination of the Constituents of German Substantive Compounds, MT 2 (1955), 3-14.

[14] J.A.Th.M. van Berckel a.o., Formal Properties of Newspaper Dutch, Tract 12 Mathematical Centre, Amsterdam 1965.

[15] J. Oosterlynck, Distributie van de fonemen in de Nederlandse één-lettergrepige monomorfematische substantieven, dissertation, Gent 1962.

Chapter 2 COMPOUND WORDS: NUMBER NAMES *

2.1. Synthesis of the number names in five languages

One of the problems in mechanical translation is the occurrence of compounds without spaces in languages such as German and Dutch; these cannot be recorded in a dictionary as any language user can produce them without limit. Little has been done on this problem. In the pioneer days of mechanical translation, E. Reifler indicated [1] how the syntactic analysis of the German compound can be done by computer. The semantic problems are almost unsurmountable.These semantic problems are quite simple for one large class of compound words: the number names. From a small number of components, around thirty, most languages can construct millions of cardinal number names. For five languages, Dutch, German, French, English and Chinese, the rules governing the construction of number names were programmed. The translation from the decimal representation into the word representation is effected by the ALGOL procedure *write number name*. Conversely the procedure *read number name* gives the translation from word form to digital form (2.2). Combination of these procedures then gives the mutual translation of number names (2.3). Recently, a number of generative grammars, most of them context-free, were published for number names in different languages [2,3,4]. Whatever the merits of these grammars, they all possess one shortcoming: the names generated have no connection with their meaning; and this while the determination of the meaning of number names is so simple. Except for extra connotations of some smaller numbers, the meaning of a number name is fully given by the decimal representation of the number referred to. Therefore we have adapted the generative grammars to permit a translation from decimal to word representation. Moreover, by treating Dutch, German,

* This chapter is a slightly amended version of a contribution to the Foundations of Language Special Volume "Grammars for Number Names" Reidel, Dordrecht 1968.

French and English in one program, a comparative grammar of the number names in these languages is found. The set of Chinese number names is generated on its own.

In the four Western languages, the number is partioned from the back in groups of three digits. The procedure *next three digits (i)* (pages 67 and 68) gives the word representation of one such group of three digits, followed by the appropriate power of ten, and then calls itself with lowered value of i, until the end of the number is reached. The procedure makes use of three other procedures:

from 1 up to 100(j, k)	producing 10 × j + k	(pages 68 and 69)
hundredfold (j, k)	producing (10×j+k)×100	(page 69)
thousand to the power (k)	producing 1000^k	(page 69)

In some languages it is possible, under certain conditions, to write a number like 1200 in two ways: as 12×100 and as 1000+200. The program investigates whether this situation exists, and, if so, gives both possibilities. (Theoretically one could perhaps write a number like 1 200 001 200 in these languages in four ways. We have used the "overlapping mode" or the "non-overlapping mode" consistently in the whole number name.) For many special measures, such as the declension of the German powers of ten and the French numbers between 70 and 99, we refer to the pages 65-69 of the program. The French word for 10^9 is given half of the time as billion, and the other half as milliard. For English we follow the Britsh system of naming 10^9 milliard and 10^{12} billion. In the American system these names are: billion and trillion. The morphemes used in the synthesis can be found in the first four columns of procedure *m* (on page 66). For each of these four languages about 33 morphemes are provided, of which some could be omitted, but at the cost of program complications.

The Chinese number name can be generated by a context-free grammar but Brainerd has shown [4] that it is better to use a class of deletion transformations which delete one component in specified contexts. These deletion transformations are readily programmable in ALGOL 60. The transformations in connection with the morpheme ling (zero) are simply

written as procedure *D(condition for context around ling, deletend)*, where, each time the Boolean *condition for context around ling* is true, the component at place *deletend* is deleted. For details we refer again to the ALGOL program (pages 70, 71, 72, 73). Besides the seven obligatory transformations, there are five optional transformations which give alternative forms. The necessary component morphemes of the Chinese number names can be seen in the last column of procedure *m* (on page 66).

2.2. Analysis of the number names in five languages

On the basis of the synthesis in section 2.1 an analysis could be given. This method, however, is:

1. Unnecessary, because many of the special measures in the procedure *write number name* are taken to ensure a correct writing of the number name, but have no informational value. In a situation where we want to read a number name, a less detailed grammar is good enough.
2. Undesirable, because we also want to read in alternative forms which are not completely correct, such as often occur for larger numbers, which are rarely written in word form. We can, e.g., neglect spacing and capitalisation, and accept rare forms like six-and-twenty.

A second method for analysis is the Reifler procedure which was described in 1.1.4. This full analysis is not necessary in our case because only in two cases (one of which is caused by the accidentally chosen transcription of the Chinese words) is an incorrect analysis arrived at if we always choose the longest possible component fitting the left-end of the name to be analysed. To simplify the program, components of only 1, 2, 3 or 4 letters are used. It appears that, with some tricks, the first four letters give enough discrimination between the components. Three numbers are associated with each component. The first number indicates the language in which the component occurs (as soon as the machine knows that it is in a certain language, only the relevant part of the dictionary is searched for the other components). The second

number indicates the number of letters the full component possesses (of which, as has been said, only four were used for identification), and the third gives the meaning of the component. These meanings are coded as follows:

code	meaning	example
0	meaningless	et, ling, the s in sechs and cents
1	1	ein, i
⋮	= code	numbers below 20
16	16	seize
20	10-fold suffix	tig, zig, uante, ty
21	10	dix
22	20	twenty
⋮	(code-20) × 10	tenfolds
24	40	quarante
30	connective between tenfolds and units	en, und
32	10^2	hundred
33	10^3	mille
⋮	$10^{code-30}$	powers of ten
42	10^{12}	trillion
51	even power of thousand suffix	on, oen
52	odd power of thousand suffix	ard
53	mil-	milli, milj
54	bil-	billi, bilj

For the analysis of the number names in five languages, a hundred components are necessary, less than for synthesis. This is because words like veertien and veertig are synthesized as a whole, but in analysis are broken down further (for analysis veer is fully synonymous

with vier, which it is not in synthesis), and because some components occur in more than one language (usually with the same meaning: acht, six. Exception: billion).

The procedure *read number name* (pages 60-64) reads the letters of the number name offered into the array *L*, thereby discarding some information which it also discarded when reading the list of component words.

The four-, three-, two-, and one-letter combinations are, in that order, compared with the entries in the component list. The meanings of the components found are stored in the array *M*. In the meantime, the tenfolds, the numbers between 10 and 20, and the powers of ten above 10^3 are combined. The notation used for Chinese morphemes makes it necessary to discriminate between pa i^4 (8×10^4) and pai (100). The optional transformation *02* obliges us to see whether ling has just been read. Also, we have to be careful not to read the English one in the German Billionen. In the languages which put the units before the tenfold (connected by "and") for the numbers below 100, this order is reversed. Except for the ambigous billion, the series of meanings in *M* is now independent of the input language. From back to front, the elements of *M* are then converted into digits in the array *N*, which will contain the number in decimal representation. To this end, a counter *pos* is kept, whose initial value is 0. The powers of ten above 10^3 give *pos* a new absolute value, and the lower powers of ten enlarge *pos* (additively). When we have reached the first element of *M*, the digital representation is completed and *number of digits* has acquired its value.

The procedure *read number name* reads all words produced by the procedure *write number name*, plus many incorrect forms. Spaces, capitalization, hyphens and apostrophes are neglected.

On the next pages the ALGOL 60 program is printed, followed by a sample output.

```
begin comment mutual translation of number names;
  integer number of digits,lang,dutch,german,french,english,chinese;
  Boolean overlap, possible overlap, o1, f, one;
  integer array W[1:108,1:4], A[0:5,0:1], N[1:16];
  comment W contains the components with their meanings. A gives the
  limits for each language in the list W. N stores the digits of
  the number translated;

  integer procedure letter;
  comment This input procedure reads the next letter of the input
  word. It neglects spaces,hyphens, apostrophes,capitalization and
  the letters q and m. It recodes the letter into the code
  a=1,...z=26. On reading a question mark the program is terminated.
  The input text is repeated,underlined,in the output;
  begin integer h;
    h:= RESYM; if h ≠ 119 then SYM(126); SYM(h); if h = 122 then EXIT;
    letter:= if h = 93 ∨ h = 120 ∨ h = 65 ∨ h = 53 ∨ h = 26 ∨ h = 49
    ∨ h = 22 then letter else if h = 4 then 22 else if h > 36 then h
    - 36 else h - 9
  end;

  procedure read number name;
  comment This procedure translates the number named by the input
  word into the decimal representation. lang gets the value of the
  input language;
  begin integer h,j,iL,iLmax,iM,iMmax,iN,w1,w2,w3,w4,m,b,pos,ling;
    integer array L[1:170], M[0:40];
    comment L stores the letters of the input word,M the meanings of
    the found components;

    procedure search(n, m); value n; integer n, m;
    comment The word n is looked up in W. If found m becomes equal
    to the index of the corresponding morfeme, else m is made
    negative. The language to which the found morfeme belongs is
    kept for future searches;
```

```
  begin integer iW;
    for iW:= A[lang,0] step 1 until A[lang,1], 1 step 1 until
    A[0,0] - 1 do
    begin if W[iW,1] = n then
      begin if lang =  0 then lang:= W[iW,2]; m:= iW; goto FOUND
      end a component is found
    end of search in relevant part of W;
    m:= - 1;
  FOUND:
  end search;

SL: for iL:= 1 step 1 until 167 do
  begin h:= letter; if h = 83 then
    begin if iL = 1 then goto SL; iLmax:= iL - 1; goto Lfilled
    end the letters of the input word are now stored in L;
    L[iL]:= h
  end;
  goto SL;
Lfilled: lang:= M[0]:= ling:= 0;
  L[iLmax + 1]:= L[iLmax + 2]:= L[iLmax + 3]:= 26; iL:= iM:= 1;
SM: if iL > iLmax then goto Mfilled; w1:= L[iL];
  w2:= w1 × 26 + L[iL + 1]; w3:= w2 × 26 + L[iL + 2];
  w4:= w3 × 26 + L[iL + 3]; search(w4, m); if m < 0 then
  begin search(w3, m); if m < 0 then
    begin search(w2, m); if m < 0 then
      begin search(w1, m); if m < 0 then goto SL end
    end
  end the first 4,3,2 and 1 letters of the input word have been
  looked up in W;
  iL:= iL + W[m,3]; b:= W[m,4];
  if lang = chinese then
  begin if b = 0 then ling:= 1 else ling:= ling + 1;
    if b = 32 ∧ L[iL] = 22 then
    begin b:= 8; iL:= iL - 1 end pa i
  end of special measures for Chinese;
```

```
if lang = english then
begin if b = 1 ∧ (M[iM - 1] = 53 ∨ M[iM - 1] = 54) then
  begin lang:= 0; b:= 51; iL:= iL - 1 end
end of special measures for English;
if b = 0 then goto SM; if b = 20 then
begin M[iM - 1]:= M[iM - 1] + 20; goto SM
end forming of tenfolds;
if b = 10 then
begin if lang = chinese then
  begin if M[iM - 1] < 10 ∧ iM ≠ 1 then
    begin M[iM - 1]:= 20 + M[iM - 1]; goto SM end
    else b:= 21
  end Chinese tenfolds;
  if lang = french then
  begin if M[iM - 1] = 26 ∨ M[iM - 1] = 28 then
    begin M[iM - 1]:= M[iM - 1] + 1; goto SM end
  end French 70 and 90
  else
  begin if M[iM - 1] < 10 ∧ M[iM - 1] > 2 then
    begin M[iM - 1]:= M[iM - 1] + 10; goto SM end
  end forming of numbers between 10 and 20
end occurrence of 10;
if lang = french then
begin if b > 6 ∧ b < 10 then
  begin if M[iM - 1] = 10 then
    begin M[iM - 1]:= 10 + b; goto SM end
  end French 17,18,19;
  if b = 22 ∧ M[iM - 1] = 4 then
  begin M[iM - 1]:= 28; goto SM
  end French 80
end of special measures for French;
if b = 51 ∨ b = 52 then
begin if M[iM - 1] = 53 then
  begin M[iM - 1]:= 30 + (if b = 51 then 6 else 9); goto SM
  end million and milliard;
```

```
      if M[iM - 1] = 54 then
      begin M[iM - 1]:= 41; goto SM
      end The ambiguous billion is given meaning 41 until we
      know whether the input language is French or not
    end;
    if lang = german then
    begin if m = 6 then goto SM
    end special measure for German sieben;
    M[iM]:= b; if iM = 40 then goto SL; iM:= iM + 1; goto SM;
Mfilled: iMmax:= iM - 1; if iMmax = 0 then goto SL;
    if lang = chinese ∧ ling ≠ 2 ∧ M[iMmax] < 10 ∧ M[iMmax - 1] ≠ 10
    then
    begin if M[iMmax - 1] = 32 then M[iMmax]:= 20 + M[iMmax] else if
      M[iMmax - 1] = 33 ∨ M[iMmax - 1] = 34 then
      begin iMmax:= iMmax + 1; M[iMmax]:= M[iMmax - 2] - 1 end
    end The effect of optional transformation O2 is reversed;
    if lang = dutch ∨ lang = german ∨ lang = english then
    begin for iM:= 2 step 1 until iMmax - 1 do
      begin if M[iM] = 30 then
        begin if M[iM - 1] < 10 ∧ M[iM + 1] > 21 ∧ M[iM + 1] < 30 then
          begin m:= M[iM - 1]; M[iM - 1]:= M[iM + 1]; M[iM]:= m end
          else M[iM]:= M[iM + 1]; iMmax:= iMmax - 1;
          for j:= iM + 1 step 1 until iMmax do M[j]:= M[j + 1]
        end
      end
    end numbers below 100 in languages where units precede tenfolds;
    for iN:= 1 step 1 until 15 do N[iN]:= 0; pos:= 0;
    for iM:= iMmax step - 1 until 1 do
    begin m:= M[iM]; if m > 31 then
      begin if m = 41 then m:= (if lang = french then 39 else 42);
        if m > 33 then pos:= m - 30 else if m = 32 then pos:= pos +
        2 else pos:= pos + (if (pos : (if lang = chinese then 4
        else 3)) × (if lang = chinese then 4 else 3) = pos then
        3 else 1); if iM = 1 ∨ m = 32 then N[15 - pos]:= 1
      end power of ten gives new value to pos
```

```
      else if m < 30 then
      begin if m < 10 then N[15 - pos]:= m else if m < 20 then
        begin N[15 - pos]:= m - (m : 10) × 10; N[14 - pos]:= 1
        end numbers below 20 are put in N
        else N[14 - pos]:= N[14-pos]+ m - 20
      end tenfolds are put in N
    end of the translation from back to front of M into N;
    for iN:= 1 step 1 until 15 do
    begin if N[iN] ≠ 0 then
      begin number of digits:= 16 - iN; goto Z end
    end determination of number of digits; goto SL; Z:
    for iN:= 1 step 1 until 15 do
    begin if iN ≤ 15 - number of digits then space else SYM(N[iN])
    end output of the decimal representation;
  end read number name;

  procedure fill W;
  comment reads in the list of components. To each component is
  assigned: its language (0=any language),its number of letters,
  and its meaning. The limits for each language are stored in A;
  begin integer h, H, i, iW, iWmax;
    iW:= 1; H:= i:= 0; iWmax:= 108; A[5,1]:= iWmax;
    lang:= 0;
  SW: h:= letter; if h = 83 then
    begin A[lang,1]:= iW - 1; lang:= lang + 1; A[lang,0]:= iW;
      goto SW
    end;
    if h ≠ 90 then
    begin i:= i + 1; if i < 5 then H:= H × 26 + h; goto SW end;
    W[iW,1]:= H; W[iW,2]:= lang; W[iW,3]:= i; W[iW,4]:= read;
    ABSFIXT(3,0,W[iW,4]); ABSFIXP(3,0,W[iW,4]);
    if iW ≠ iWmax then
    begin iW:= iW + 1; H:= i:= 0; goto SW end;
    A[0,0]:= A[1,0]; A[0,1]:= iWmax
  end fill W;
```

```
procedure write(s); string s;
comment This output procedure writes the string s;
begin PRINTTEXT(s); PUTEXT(s) end;

procedure write number name(N, lengthN, lang); array N;
integer lengthN, lang;
comment The number in array N with length lengthN is written in
the language lang;
begin integer i;
  if lang ≠ 1 then NL;
  for i:= 1 step 1 until (if lang = 1 then 3 else 18) do space;
  if lang = chinese then CHINESE(N) else
  begin possible overlap:= overlap:= false;
    Next 3 digits((lengthN - 1) : 3); if possible overlap then
    begin overlap:= true; NL; write(‡or with overlap: ‡);
      Next 3 digits((lengthN - 1) : 3);
    end overlapping case
  end non chinese language
end write number name;

procedure m(j, k); value j, k; integer j, k;
comment The word in row 9×j+k and the column of the output
language is written;
begin switch morfemes:= m1, m2, m3, m4, m5, m6, m7, m8, m9, m11,
  m12, m13, m14, m15, m16, m17, m18, m19, m10, m20, m30, m40,
  m50, m60, m70, m80, m90, p2, p3, p6or4, p9or8, p12, m0;
  procedure P(du, ge, fr, en, ch); string du, ge, fr, en, ch;
  begin if lang = dutch then write(du) else if lang = german then
    write(ge) else if lang = french then write(fr) else if lang =
    english then write(en) else write(ch); goto WRITTEN
  end P;
  procedure Q(du, ge, fr, en); string du, ge, fr, en;
  P(du, ge, fr, en, ‡ ‡);
  procedure R(s); string s; P(s, s, s, s, s);
  goto morfemes[9 × j + k];
```

```
comment
        Dutch           German          French          English         Chinese;
m 1:P( ‹een›,          ‹ein›,          ‹un›,           ‹one›,          ‹i›);
m 2: if o1 then begin o1:=false;R(
    P( ‹twee›,         ‹zwei›,         ‹deux›,         ‹two›,          ‹erh›);
m 3:P( ‹drie›,         ‹drei›,         ‹trois›,        ‹three›,        ‹san›);
m 4:P( ‹vier›,         ‹vier›,         ‹quatre›,       ‹four›,         ‹ssu›);
m 5:P( ‹vijf›,         ‹fuenf›,        ‹cinq›,         ‹five›,         ‹wu›);
m 6:P( ‹zes›,          ‹sechs›,        ‹six›,          ‹six›,          ‹liu›);
m 7:P( ‹zeven›,        ‹sieben›,       ‹sept›,         ‹seven›,        ‹ch'i›);
m 8:P( ‹acht›,         ‹acht›,         ‹huit›,         ‹eight›,        ‹pa›);
m 9:P( ‹negen›,        ‹neun›,         ‹neuf›,         ‹nine›,         ‹chiu›);
m11:Q( ‹elf›,          ‹elf›,          ‹onze›,         ‹eleven›);
m12:Q( ‹twaalf›,       ‹zwoelf›,       ‹douze›,        ‹twelve›);
m13:Q( ‹dertien›,      ‹dreizehn›,     ‹treize›,       ‹thirteen›);
m14:Q( ‹veertien›,     ‹vierzehn›,     ‹quatorze›,     ‹fourteen›);
m15:Q( ‹vijftien›,     ‹fuenfzehn›,    ‹quinze›,       ‹fifteen›);
m16:Q( ‹zestien›,      ‹sechzehn›,     ‹seize›,        ‹sixteen›);
m17:Q( ‹zeventien›,    ‹siebzehn›,     ‹dix-sept›,     ‹seventeen›);
m18:Q( ‹achttien›,     ‹achtzehn›,     ‹dix-huit›,     ‹eighteen›);
m19:Q( ‹negentien›,    ‹neunzehn›,     ‹dix-neuf›,     ‹nineteen›);
m10:P( ‹tien›,         ‹zehn›,         ‹dix›,          ‹ten›,          ‹shih›);
m20:Q( ‹twintig›,      ‹zwanzig›,      ‹vingt›,        ‹twenty›);
m30:Q( ‹dertig›,       ‹dreissig›,     ‹trente›,       ‹thirty›);
m40:Q( ‹veertig›,      ‹vierzig›,      ‹quarante›,     ‹forty›);
m50:Q( ‹vijftig›,      ‹fuenfzig›,     ‹cinquante›,    ‹fifty›);
m60:Q( ‹zestig›,       ‹sechzig›,      ‹soixante›,     ‹sixty›);
m70:Q( ‹zeventig›,     ‹siebzig›,      ‹soixante›,     ‹seventy›);
m80:Q( ‹tachtig›,      ‹achtzig›,      ‹quatre-vingt›, ‹eighty›);
m90:Q( ‹negentig›,     ‹neunzig›,      ‹quatre-vingt›, ‹ninety›);
p2 :P( ‹honderd›,      ‹hundert›,      ‹ cent›,        ‹ hundred ›,    ‹pai›);
p3 :P( ‹duizend›,      ‹tausend›,      ‹ mille ›,      ‹ thousand ›,   ‹ch'ien›);
p6or4:P(‹miljoen›,     ‹ Million›,     ‹ million›,     ‹ million›,     ‹wan›);
p9or8: f:=¬f; if fΛlang=french then R(  ‹ billion›) else
    P( ‹miljard›,      ‹ Milliarde›,   ‹ milliard›,    ‹ milliard›,    ‹i4›);
p12:P( ‹biljoen›,      ‹ Billion›,     ‹ trillion›,    ‹ billion›,     ‹chao›);
m 0:P( ‹en›,           ‹und›,          ‹ et ›,         ‹ ›,            ‹ling›);
```

```
  WRITTEN:
  end m(j,k);

  procedure and; m(0, 33);
  procedure space; write(↓ ↑);
  procedure hyphen; write(↓-↑);
  procedure SYM(n); integer n;
  comment If n is a digit, it is written by this output procedure;
  begin PRSYM(n); PUSYM(n) end;
  procedure NL; write(↓
↑);

  procedure Next 3 digits(i); integer i;
  comment The i-th group from the back of three digits is
  produced,followed by an appropriate power of thousand. It then
  calls itself with lowered i until i becomes 1;
  begin integer i3;
    i3:= i × 3; if ¬overlap then
    begin if ¬possible overlap then possible overlap:= i ≠ 0 ∧ N[13 -
      i3] = 0 ∧ N[14 - i3] = 0 ∧ N[15 - i3] ≠ 0 ∧ (lang = french ¬
      N[15 - i3] = 1) ∧ N[16 - i3] ≠ 0;
    NON OVERLAP: hundredfold(0, N[13 - i3]);
      if N[13 - i3] ≠ 0 then
      begin if lang = french then
        begin if N[14 - i3] + N[15 - i3] = 0 ∧ N[13 - i3] ≠ 1 then
          write(↓s↑); space
        end cents
      end first of the 3 digits;
      from 1 up to 100(N[14 - i3], N[15 - i3]);
      if i = 0 ∧ lang = german ∧ N[14] = 0 ∧ N[15] = 1 then write(↓s↑);
      one:= (N[13 - i3] = 0 ∧ N[14 - i3] = 0 ∧ N[15 - i3] = 1);
      if i = 0 then goto MADE;
      if N[13 - i3] + N[14 - i3] + N[15 - i3] ≠ 0 then
      thousand to the power(i); Next 3 digits(i - 1)
    end of non overlapping case
```

```
    else
    begin if i ≠ 0 ∧ N[13 - i3] = 0 ∧ N[14 - i3] = 0 ∧ N[15 - i3] ≠ 0
      ∧ (lang = french ¬ N[15 - i3] = 1) ∧ N[16 - i3] ≠ 0 then
      begin hundredfold(N[15 - i3], N[16 - i3]);
        if lang = french then
        begin if N[17 - i3] + N[18 - i3] = 0 then write(‹s›); space
        end cents ;
        from1upto100(N[17 - i3], N[18 - i3]);
        if i = 1 ∧ N[14] = 0 ∧ N[15] = 1 ∧ lang = german then write(
        ‹s›); one:= false; if i = 1 then goto MADE;
        thousand to the power(i - 1); Next 3 digits(i - 2)
      end overlapping case
      else goto NON OVERLAP
    end;
MADE:
end Next 3 digits(i);

procedure from 1 up to 100(j, k); value j, k; integer j, k;
comment produces 10×j+k;
if k ≠ 0 then
begin if j < 2 then m(j, k) else
  begin if lang = french ∨ lang = english then
    begin m(2, j); if lang = english then
      begin hyphen; m(0, k) end
      else
      begin if j ≠ 7 ∧ j ≠ 9 then
        begin if k = 1 then
          begin if j ≠ 8 then and end
          else hyphen; m(0, k)
        end
        else
        begin if k = 1 ∧ j = 7 then and else hyphen;
          from 1 up to 100(1, k)
        end
```

```
      end French up to 100
    end French and English
    else
    begin m(0, k); and; m(2, j) end Dutch and German
  end above 20
end
else
begin if j ≠ 0 then
  begin m(2, j); if lang = french ∧ j > 6 then
    begin if j = 8 then write(‹s›) else
      begin hyphen; from 1 up to 100(1, 0) end
    end French 70,80,90
  end tenfolds
end 1 up to 100;

procedure hundredfold(j, k); value j, k; integer j, k;
comment produces (10×j+k)×100;
begin if j + k ≠ 1 ∨ lang = english then from 1 up to 100(j, k);
  if k ≠ 0 then m(3, 1)
end hundredfold;

procedure thousand to the power(k); value k; integer k;
comment produces 1000↑k;
begin if k > 1 then
  begin if one ∧ lang = german then write(‹e›) end;
  m(3, k + 1); if k > 1 then
  begin if ¬one then
    begin if lang = french then write(‹s›)
      else if lang = german then
      begin if k ≠ 3 then write(‹e›); write(‹n›) end
    end;
    if lang ≠ dutch then space
  end
end 1000↑;
```

```
procedure CHINESE(Number); array Number;
comment The name for the number in array Number is produced;
begin integer i, shih, pai, chien, wan, i4, chao, j, L, LL;
  Boolean change;
  integer array S[ - 1:36];
  comment The string of Chinese morfemes with length L is stored
  in S, numbers below 10 in their natural code,powers of ten as
  10, 11, 12, 13, 14, 15 (chao);

  procedure write chinese;
  comment The string of Chinese morfemes in S is produced;
  for j:= 1 step 1 until L do
  begin if S[j] = 0 then and else if S[j] < 10 then m(0, S[j])
    else if S[j] = 10 then m(0, 19) else m(2, S[j] - 1); space
  end;

  comment The 7 obligatory transformations: ;

  procedure D(condition for context of ling, deletend);
  Boolean condition for context of ling; integer deletend;
  for j:= 1 step 1 until L do
  begin if ling(j) then
    begin if condition for context of ling then delete(deletend) end
  end general deletion transformation around ling;

  procedure delete(n); value n; integer n;
  begin integer k;
    L:= L - 1;
    for k:= n step 1 until L do S[k]:= S[k + 1]; change:= true
  end delete ;

  procedure D1; D(shih pai chien(j - 1) ∧ wan i4 chao(j + 1), j);

  procedure D2;
  D((shih pai chien(j - 1) ∨ wan i4 chao(j - 1))
  ∧ shih pai chien(j + 1), j + 1);
```

```
procedure D3; D(ling(j + 1), j + 1);

procedure D4; D(chao i4(j - 1) ∧ i4 wan(j + 1), j + 1);

procedure D5;
if ling(1) ∧ L > 2 ∧ (shih pai chien(2) ∨ wan i4 chao(2)) then
begin delete(1); delete(1) end;

procedure D6; if ling(L) ∧ L > 1 then delete(L);

procedure D7; if S[1] = i ∧ S[2] = shih then delete(1);

comment The 5 optional transformations: ;

procedure option(n); integer n;
comment if the application of an optional transformation
is possible the alternative form is produced;
begin NL; write(‡or with option o‡); SYM(n); write(‡:‡);
  write chinese
end option;

procedure O1;
if S[1] = 2 ∧ (wan i4 chao(2) ∨ (shih pai chien(2)) ∧ S[2] ≠ shih)
then
begin o1:= true; option(1) end;

procedure O2;
begin integer a, b, c;
  a:= S[L - 2]; b:= S[L - 1]; c:= S[L];
  if ((a = chien ∧ c = pai) ∨ (a = pai ∧ c = shih) ∨ (a = wan ∧
  c = chien)) ∧ b > 0 ∧ b < 10 then
  begin delete(L); option(2) end
end O2 ;
```

```
procedure O3;
for j:= 2 step 1 until L - 1 do
begin if S[j] = shih then
  begin if S[j - 1] = i ∧ S[j + 1] > 0 ∧ S[j + 1] < 10 then
    begin delete(j - 1); option(3); O3 end
  end
end;

procedure O4;
begin change:= false;
  D(S[j + 1] < 10 ∧ S[j + 2] = pai ∧ ling(j + 3) ∧ S[j + 4] <
  10, j); if change then option(4)
end O4;

procedure O5;
begin integer jj;
  change:= false;
  for j:= 2 step 1 until L - 4 do
  begin if chaoi4(j) then
    begin if ling(j + 1) then
      begin for jj:= 2 step 1 until j - 1 do
        begin if chaoi4(jj) ∨ ling(jj) then goto OUT end;
        for jj:= L step - 1 until j + 3 do
        begin if i4wan(jj) then goto OUT; if ling(jj) then
          begin delete(j + 1); goto OUT end
        end investigation of string to the right of ling
      end chao i4 followed by ling
    end occurrence of chao i4;
  OUT:
  end investigation of string;
  if change then option(5)
end O5;
```

```
    Boolean procedure ling(n); integer n; ling:= S[n] = 0;
    Boolean procedure shih pai chien(n); integer n;
    shih pai chien:= S[n] ≥ shih ∧ S[n] ≤ chien;
    Boolean procedure wan i4 chao(n); integer n;
    wan i4 chao:= S[n] ≥ wan;
    Boolean procedure chao i4(n); integer n;
    chao i4:= S[n] ≥ i4;
    Boolean procedure i4 wan(n); integer n;
    i4 wan:= S[n] = i4 ∨ S[n] = wan;

    i:=1; shih:=10; pai:=11; chien:=12; wan:=13; i4:=14; chao:=15;
    for j:= - 1, 0, 32, 33, 34, 35, 36 do S[j]:= - 1; S[1]:= 0;
    for j:= 3 step 2 until 31 do S[j]:= Number[(j) : 2];
    for j:= 2 step 8 until 26 do
    begin S[j]:= chien; S[j + 2]:= pai; S[j + 4]:= shih end;
    S[8]:= chao; S[16]:= i4; S[24]:= wan; L:= 31;
  leading lings: change:= false; D5;
    if change then goto leading lings;
  TRANSFORM: change:= false; LL:= L; D(true, 32); if change then
    begin L:= LL; change:= false; D1; D2; D3; D4; D6;
      if change then goto TRANSFORM
    end obligational transformations in connection with ling;
    D7; write chinese; O1; O2; O3; O4; O5;
  end chinese number
```

With the above procedures translation programs of many kinds can be written. One of them, reading in any language and translating into all five languages, is:;

```
  o1:= f:= false; fillW; lang:= 0;
  dutch:= 1; german:= 2; french:= 3; english:= 4; chinese:= 5;
START: NL; read number name;
  for lang:= dutch, german, french, english, chinese do
  write number name(N, number of digits, lang); goto START
end
```

acht 8 ard 52 billi 54 e 0 elf 11 en 30 hundred 32 milli 53
on 51 s 0 six 6 vier 4
bilj 54 der 3 drie 3 duizend 33 een 1 honderd 32 milj 53
negen 9 oen 51 tachtig 28 tien 10 tig 20 twaalf 12 twee
2 twintig 22 veer 4 vijf 5 zes 6 zeven 7
drei 3 ein 1 fuenf 5 funf 5 neun 9 sech 6 sieb 7 ssig
20 tausend 33 und 30 zehn 10 zig 20 zwanzig 22 zwei 2 zwolf
12 zwoelf 12
cent 32 cinq 5 deux 2 dix 10 douze 12 et 0 huit 8 mille
33 neuf 9 onze 11 seize 16 sept 7 soixante 26 treize 13
trente 23 trillion 42 trois 3 quante 20 quatre 4 quatorze
14 quarante 24 quinze 15 un 1 vingt 22
and 30 eigh 8 eleven 11 fif 5 five 5 for 4 four 4 nine
9 one 1 seven 7 t 0 teen 10 ten 21 thir 3 thousand 33
three 3 twelve 12 12 twenty 22 two 2 ty 20
chao 42 ch'i 7 ch'ien 33 chiu 9 erh 2 i 1 i4 38 liang
2 ling 0 liu 6 pa 8 pai 32 san 3 shih 10 ssu 4 wan
34 wu 5

eenmiljard tweehonderd miljoen zeshonderdduizend vijfhonderd
1200600500 eenmiljardtweehonderdmiljoenzeshonderdduizendvijfhonderd
or with overlap: twaalfhonderdmiljoenzeshonderdduizendvijfhonderd
eine Milliarde zweihundert Millionen sechshundert tausendfuenfhundert
or with overlap: zwoelfhundert Millionen sechshunderttausendfuenfhundert
un billion deux cents millions six cents mille cinq cents
or with overlap: douze cents millions six cents mille cinq cents
one milliard two hundred million six hundred thousand five hundred
or with overlap: twelve hundred million six hundred thousand five hundred
shih erh i4 ling liu shih wan ling wu pai
or with option o5: shih erh i4 liu shih wan ling wu pai
pa i4 ling i pai i shih pa
800000118 achthonderdmiljoenhonderdachttien
achthundert Millionen hundertachtzehn
huit cents millions cent dix-huit
eight hundred million one hundred eighteen
pa i4 ling i pai i shih pa
or with option o3: pa i4 ling i pai shih pa

seven billion
7000000000000 zevenbiljoen
sieben Billionen
sept trillions
seven billion
ch'i chao

sept billion
7000000000 zevenmiljard
sieben Milliarden
sept billions
seven milliard
ch'i shih i4

siebenhundertsiebenundsiebzig
777 zevenhonderdzevenenzeventig
siebenhundertsiebenundsiebzig
sept cent soixante-dix-sept
seven hundred seventy-seven
ch'i pai ch'i shih ch'i

pa pai chiu
890 achthonderdnegentig
achthundertneunzig
huit cent quatre-vingt-dix
eight hundred ninety
pa pai chiu shih

or with option o2:pa pai chiu

five-and-twenty
25 vijfentwintig
fuenfundzwanzig
vingt-cinq
twenty-five
erh shih wu

2.3. Translation of number names

It is clear how the two procedures *write number name* and *read number name* make mutual translation in the five languages possible. In the case where input language and output language are the same, this translation can be interpreted as: " give the correct version of the number name plus all its alternative forms".
The program reproduced on pages 60-73 gives, for every number name read, its translation in the five languages. In the output (on pages 74-75), the input is reproduced with underlinings, including the component dictionary. The runtime on the X8 was 5 seconds.
The maximal number processed is $10^{15}-1$ but extension is simple (for Dutch, the synthesis up to $10^{66}-1$ was done in [5]). In such an extension the difference between the American and the English systems of denominations above one million becomes more pronounced.
In the case of number name translation between several languages, the use of the decimal representation as an intermediate language seems obvious. The use of an intermediate language for the mutual translation of several languages in general may, however, very well be impossible or undesirable.

Sources Chapter 2

[1] E. Reifler, Mechanical Determination of the Constituents of German Substantive Compounds, MT 2 (1955), 3-14.

[2] H. Brandt Corstius (ed), Grammars for Number Names, Reidel, Dordrecht 1968.

[3] R.P.G. de Rijk, Une grammaire "context-free" pour la génération mécanique des noms de nombres français, in: Braffort and van Schepen, Automation in language translation etc., Euratom CID, Brussels 1967.

[4] B. Brainerd, Two Grammars for Chinese Number Names, Canadian Journal of Linguistics, 12 (1966), 33-51.

[5] H. Brandt Corstius, Automatic Translation of Numbers into Dutch, Foundations of Language 1 (1965), 39-62.

Chapter 3 WORD GROUPS : NOUN PHRASES

3.1. Isolation of noun phrases in Dutch sentences

3.1.1. Aim

One of the reasons that no adequate generative grammar has been constructed for any natural language is that it is hard to divide the problem into meaningful subproblems. We seldom have the ideal situation, as was the case in the preceding chapter, of an isolated subgrammar which can be constructed independently of the full grammar of the language. Even in the case of the number names one could argue that, e.g., the declension of the German names for powers of ten should not be treated in the number name grammar but in a more general grammar concerning German nouns.
Broadly speaking, a generative grammar of Dutch would contain three parts: the first part constructs noun phrases that can act as, e.g., subject or object, the second part concerns the verb and constructs sentences with one finite verb form, and the third part constructs compound sentences from these simple sentences. These three parts are, however, intertwined. It is our aim to isolate as far as possible the first part from the rest of the grammar. We will indicate a way to find maximal noun phrases in Dutch sentences without supposing anything about the rest of the grammar in which this part will fit. Our proposed noun phrase grammar as such, will certainly not be part of the full grammar of the language. To give an example: it is customary to derive the adjective-substantive combination with the aid of a transformation from the deep structure underlying a sentence with to be, e.g.,

rozen zijn rose → rose rozen.

This enables one to give the selection restrictions only once for the verb zijn. We will, however, generate this construction by rules like:

noun → (adjective) noun, rozen
adjective → rose

In the same way, participial constructions will normally be made by transformations from verbal constructions, whereas we introduce the participles as a special kind of adjectives.
If we know that our proposed noun phrase grammar will not, as such, be part of the full grammar, what is its use? We hope to attain the following:

1. To show that it is possible, by using a parsing algorithm based on a context-free grammar, to isolate most noun phrases in Dutch sentences (section 3.1).
2. To acquire insight in the structure of these noun phrases, notably in the quantitative aspects: how often is such and such a rule applied? (section 3.2).
3. To acquire insight in the structure of Dutch sentences in which noun phrases are condensed into the determiner (e.g., the article) and the central substantive. The remaining sentence is still grammatical but much simpler, and can be studied to form hypotheses about the other two parts of the grammar (section 3.3). To give a simple example: the hypothesis: "the subject of a sentence is the first noun phrase, not preceded by a preposition" can only be tested if the term "noun phrase" is rigidly defined.

Apart from these syntactical considerations the isolation of noun phrases from sentences may have its use for language statistics, computational stylistics, readability scores, information retrieval, etc.

We give the definition of the Dutch noun phrase in 3.1.2, and discuss its shortcomings and the question of hearer-speaker grammars in 3.1.3. A program that uses this grammar is explained in 3.1.4 and we show a sample output in 3.1.5.

3.1.2. Definition of the Dutch noun phrase

A preliminary definition of the Dutch noun phrase in terms of substitutability is: any sequence of words not containing a finite verb form which can be put in the place of the word iets in the sentence ik denk aan iets such that the resulting string will be a grammatical sentence. We

have to exclude finite verb forms, in the first place for the trivial reason that the resulting sentence may not become a coordination of two sentences, and in the second place because noun phrases containing a whole sentence, however frequent they are, cannot be treated as long as we do not possess a full grammar. From now on, if we say "noun phrase", we mean "noun phrase without finite verb form". The preliminary definition entails that noun phrases are continuous. Indeed, this seems to be a fair assumption, because in cases like: ik heb een tafel nodig met drie poten the discontinuity in the noun phrase will, if we accept discontinuous noun phrases at all, be formed by a transformation dependent on the verb nodig hebben.
By this definition, only the noun phrases in the accusative are found, but in Dutch this case only influences the form of pronouns, which are excluded.

It is clear that this preliminary definition is useless because it employs a concept like "grammatical sentence" which it is supposed to define in its turn. We only gave it to present an informal description of the concept "noun phrase" that finds its real definition in the context-free grammar in table 14 on page 82. This grammar needs little explanation: one starts with the symbol *N* and applies the rewriting rules until only terminal elements, i.e., Dutch words, remain in the string. The word classes, i.e., those nonterminal elements that are only rewritten in terminal elements, have been given names that suggest known grammatical categories but these names have no significance in themselves.
Higher categories are written in italic letters, sometimes with a number.
Five rules, only differing in a number, are condensed into two rules.
The grammar leans heavily on the study by Van der Lubbe [1] and on the texts that we used to test it out [2].

3.1.3. Grammars for hearers and for speakers

We want to isolate maximal noun phrases from Dutch text. If we have a synthetic grammar, in a context-free form, for the Dutch noun phrase,

TABLE 14 : Noun phrase grammar

N → zowel *N* als *N*, *n*(coordinator *N*)

n → *FACANTE SUBSPHRASE FACPOST*

FACANTE → (*ante4*) (*ante3*) (*ANTE2*) (*ANTE1*)

FACPOST → (*POST1*) (*POST2*) (*POST3*) (*post4*)

ANTEi → *antei* (coordinator *ANTEi*), zowel *ANTEi* als *ANTEi* i=*1,2*

POSTi → *posti* (coordinator *POSTi*), zowel *POSTi* als *POSTi* i=*1,2,3*

ante4 → anteparticle

ante3 → determiner, al *ante3*

ante2 → numeral, advdegree *ante2*

ante1 → *adjphrase*, *infphrase*, *partphrase*

SUBSPHRASE → *subsphrase* (coordinator *SUBSPHRASE*)

subsphrase → contsubst *FACPOST SUBSPHRASE*, substantive

post1 → determgenitive *N*, (determiner) numeral

post2 → postadverb, *prepphrase*

post3 → om *infphrase*

post4 → postparticle

adjphrase → adjective,quantsubst quantadj,zo adjective mogelijk, adverb *adjphrase*,*prepphrase adjphrase*,adjective *adjphrase*

infphrase → te infinitive, adverb *infphrase*,*prepphrase infphrase*, preposition te infinitive

partphrase → participle,adverb *partphrase*,*prepphrase partphrase*

prepphrase → preposition *N*, prepadverb *prepphrase*, anteparticle *prepphrase*

zowel	→ zowel, hetzij ,...		postadverb	→ hier, daar	,...
als	→ als, hetzij ,...		postparticle	→ zelfs, alleen	,...
coordinator	→ en, of ,...		adjective	→ mooi, mooie	,...
anteparticle	→ zelfs, ook ,...		quantsubst	→ meters, kilo	,...
determiner	→ de, het ,...		quantadj	→ lange, zware	,...
numeral	→ twee, vele ,...		adverb	→ nogal, allerminst	,...
advdegree	→ zeer, tamelijk,...		infinitive	→ geven, nemen	,...
contsubst	→ kist, minister,...		participle	→ gevend, gegeven	,...
substantive	→ man, paard ,...		preposition	→ in, van	,...
determgenitive	→ der, des ,...		prepadverb	→ vlak, juist	,...

then it is well known that such a grammar can be used for analysis. It enables us to answer two questions concerning any word string offered:

1. Is the word string a grammatical noun phrase?
2. If so, what is its structure?

We are, however, not so much interested in these questions but in the question: what are the maximal noun phrases in a Dutch text, of which the grammaticalness is taken for granted? Therefore, a coarser grammar is sufficient for our purpose; in this sense: the grammar must generate all noun phrases that we want to recognize in a text, but it may also generate other strings as long as these strings do not occur in a Dutch text. The intersection of the language generated by our grammar and the set of word strings in a Dutch text has to be the set of noun phrases. In this stage of syntactic investigation it seems permissible to make use of the information contained in the fact that a certain text is grammatical.

In reviewing the shortcomings of the proposed grammar, we first call attention to cases where the grammar fails for analysis: noun phrases (according to our preliminary definition) which are not formed by our generative grammar. There seem to be only a few of these cases, e.g., een man gewend te bevelen (with uncommon position of the present participle), and een wandeling het bos in (with uncommon position of the preposition).

Much more frequent are the cases where the grammar fails for synthesis: nongrammatical noun phrases (according to the preliminary definition) are formed by the generative grammar. This is caused by the fact that no differentiation is made between de-words (male and female) and het-words (neuter), and between singular and plural. Applications of the rewriting rules would, e.g., produce: de mooi paard instead of het mooie paard. It is certainly possible to remedy this situation by generating de-nouns and het-nouns, plurals and singulars, separately, but for our aim, i.e. for analysis, this seemed unnecessary: the sequence de mooi paard will simply not appear in the texts used.

One shortcoming is that constructions like zowel N als N or hetzij N hetzij N are not properly defined by our rules, since these rules also produce nongrammatical constructions like zowel N hetzij N, but, again, these will not occur in the texts to be analyzed.
So far, we have only discussed surface structure. Indeed, we are already satisfied if we isolate the maximal noun phrase de stoelen van de zolder van Jan from the sentence: ik denk aan de stoelen van de zolder van Jan without knowing whether its structure is (de stoelen van de zolder) van Jan or de stoelen van (de zolder van Jan). The only demand that we make on structure is that when the noun phrase is replaced by its components *ante3 SUBSPHRASE*, the resulting sentence remains grammatical.

3.1.4. Method of the program for noun-phrase isolation

Several methods exist to construct parsing programs for specific context-free grammars [3]. We choose the method of C.H.A. Koster [4] which can be characterized as a syntax-directed top-to-bottom analyzer.
A parsing algorithm is called "syntax-directed" if it is a general process which takes the specific syntax as parameter. Thus, a linguist need not read or understand the parsing program, but he can work with the rules themselves.
The parsing method used is syntax-directed in the sense that from a given syntax a parsing program in ALGOL 60 is obtained by a formal process of transcription.
A parsing algorithm is called "top-to-bottom" if it works by tentative generation from the initial symbol of the grammar downwards. In our method, starting from a call of the procedure *noun phrase*, a hierarchy of procedures corresponding to ever lower nodes in the phrase marker is called.
The parsing program consists of two parts, the standard part and the particular part, which is dictated by the grammar used.
The standard part, which does not depend on the particular grammar used, contains a number of declarations and statements taking care of the

administrative work of the parser. It contains declarations for an input array *I* containing the sequence of symbols to be parsed, an input pointer *pin* pointing to the symbol presently under consideration and Boolean procedures *match* and *END* with declarations reading, apart from details:

```
Boolean procedure match(word,Q); integer word; Boolean Q;
if I[pin] = end marker then match:= false else
if I[pin] = word then
begin pin:= pin + 1; match:= Q; pin:=pin - 1 end
else match:= false;
Boolean procedure END;
END:= I[pin] = end marker;
```

The particular part is obtained from the grammar by a process of transcription: Every rule of the grammar is transcribed into the declaration of a Boolean procedure recognizing terminal productions of a specific nonterminal.
Thus the mutually recursive rules of the grammar become a set of mutually recursive procedures in the parsing program. Each of these procedures is equipped with a Boolean parameter *Q*, which is called by name.
How the transcription is made, and how the program analyzes strings in *I* is illustrated by an example:
Suppose the grammar has only two terminal elements: zeer and oud. The nonterminals are: 'adverb', 'adjective' and the initial symbol '*adjphrase*'. There are three production rules:

adjphrase → adverb^*adjphrase*, adjective
adverb → zeer
adjective → oud

The generated adjphrases thus consist of the adjective oud preceded by a string of 0,1, or more adverbs zeer.

The rules are transcribed into the following ALGOL procedures:

```
Boolean procedure adjphrase(Q); Boolean Q;
        adjphrase:= adverb(adjphrase(Q)) ∨ adjective(Q);
Boolean procedure adverb(Q); Boolean Q;
        adverb:= match(zeer,Q);
Boolean procedure adjective(Q); Boolean Q;
        adjective:= match(oud,Q);
```

where *zeer* and *oud* are integers with values corresponding to the codes the words zeer and oud are given in the input array *I*. In combination with the ALGOL procedures *match* and *END* that were declared above, we only have to make the call: *adjphrase (END)* to analyze the contents of *I*. Suppose *I* contains only the word oud and an endmarker. Then the call *adjphrase (END)* will first evaluate *adverb (adjphrase (END))* and return with the value *false*. Then the second part of the disjunction: *adjective (END)* is evaluated by calling *match (oud, END)* which gets the value *true*. This true-value is given upwards to *adjective* and to *adjphrase:* one parsing has been found. If more than one alternative in the procedure body of a nonterminal is true we have an ambiguity giving more than one parsing of the nonterminal in question.
If one is interested only in obtaining at most one parsing without being bothered by ambiguity, then instead of the or-symbol one may use in the parsing procedures a construction with *if ... then true else* which ensures that if the first alternative is *true* the others are not investigated.

This method works for all context-free grammars without left-circularity, i.e., without rules that permit a derivation A => Aω, where A is a non-terminal and ω an arbitrary string. There is an algorithm to determine

whether a context-free grammar is non-left-circular [5].

The newspaper text we used consists of sentences, so even though we are interested only in the noun phrases, we have to extend the syntax of *noun phrase* to that of a rather sketchy *sentence*, by adding the rule:

sentence → *noun phrase* ^ *sentence*, *singleword* ^ *sentence*

Ordering the alternatives in the noun-phrase grammar in such a way that the longest and most involved alternative is tried first, the parse yielded by our program displays the maximal noun phrase from a left starting point. Initially, this left starting point is the first word of the sentence, later on it is the first word following the last noun phrase isolated. Note that in this way not all the maximal noun phrases in the text need to be found: it is possible that by shortening the left one of a pair of adjoining noun phrases, the right one becomes longer, and maximal. The preference for the left seems warranted in written Dutch.

In the next section, the program described is applied to a sample of newspaper prose.

< italiaanse +2+ kersen >
< +1+ ons +2+ artikel > < +2+ moskou > < +2+ belgrado >
< +2+ rome > behoeft < +1+ een +2+ vervolg > zoals dit trouwens met
< alle +2+ artikelen in < +1+ deze +2+ rubriek > > < +1+ het +2+
geval > is want < +1+ de +2+ geschiedenis van < +2+ vandaag > > zet
zich morgen voort
maar met < +1+ genoemd +2+ artikel > is het wel < +1+ een bijzonder
+2+ geval > daar < +1+ de derde +2+ schakel van < +1+ de +2+ reeks
> > moskou-belgrado-rome thans < +2+ zeer > duidelijk voor < +1+ het
+2+ vizier > komt met < +1+ de recente +2+ woorden van < +2+ togliatti > >
< +1+ de +2+ leider van < +1+ de italiaanse +2+ communisten > +2+
palmiro togliatti > heeft bekend dat < +1+ zijn +2+ vertrouwen in < +1+
het +2+ kremlin > > geschokt is
hij revolteert tegen < +2+ moskou >
hij vraagt om < +2+ opheldering >
hij rekent < +1+ het +2+ chroestsjew > cs wel toe dat zij tijdens < +1+
het +2+ leven van < +2+ stalin > > inactief zijn gebleven als deze < +1+
een zo monsterlijke +2+ tiran > is geweest als deze op < +1+ het twintigste
+2+ congres te < +2+ moskou > > is afgeschilderd
maar < +2+ stalin > was ook niet zo slecht zegt < +2+ togliatti > als
hij nu afgeschilderd wordt hij had < ook +1+ zijn +2+ verdiensten voor <
+1+ het +2+ communisme of +2+ socialisme > > zoals het thans bij
< +2+ voorkeur > wordt genoemd
men zal wat < +2+ stalin > betreft < +1+ het door < +2+ moskou >
geschilderde +2+ beeld > dienen bij te schilderen en van < +1+ de huidige
+2+ kremlinheersers > zal men wat < +2+ deugd > af moeten tellen en
men zal zo op < +1+ een +2+ middenkoers > uit moeten komen hetgeen
dan uiteraard inhoudt < +1+ een onafhankelijke +2+ koers van < +2+
moskou > > < +1+ een +2+ vorm van < nationaal-communistische +2+
beweging > > < +1+ een +2+ divergentie van < +1+ het +2+ kremlin
als < +2+ dirigent van < +1+ het internationale +2+ communisme > > > >

< nadere +2+ beschouwing >
< +1+ de +2+ uitslag van < +1+ de +2+ verkiezingen > > is niet < +1+ een +2+ zaak > waaraan men < +1+ een +2+ artikeltje > wijdt om dan vervolgens maar < +2+ weer > spoedig tot < +1+ de +2+ orde van < +1+ de +2+ dag > > over te gaan
er is < +1+ een gevoelige +2+ nederlaag > geleden
en het is dus nu < +2+ zaak om < +2+ ons over < +1+ de +2+ aard > en < +1+ de +2+ oorzaken > > > daarvan te beraden en te zien wat daaraan gedaan kan worden
want in XXgetal is er < +2+ weer > < +1+ een +2+ verkiezingsstrijd > en < vier +2+ jaren > duren maar kort
< +1+ de +2+ aard van < +1+ de +2+ nederlaag > > het is moeilijk om tot < +1+ een goede +2+ analyse van < +1+ de +2+ stemmenverschuiving > > te komen
want dat is altijd < +1+ een gecompliceerd +2+ geval >
maar het is duidelijk dat er om < +2+ sprake > is van < +1+ een opvallende +2+ verschuiving > van stemmen van < +1+ de prot chr +2+ partijen naar < +1+ de +2+ pvda > >
< +1+ de +2+ oorzaak > van die < +2+ verschuiving > is eveneens gecompliceerd
er zullen < +2+ mensen > bij betrokken zijn die principiee' voor < +1+ de +2+ doorbraak > hebben gekozen
die bewust op < theologische en politieke +2+ gronden > < +1+ de +2+ idee der < christelijke +2+ partijformatie > > hebben vaarwelgezegd
maar < +1+ het +2+ merendeel > zal < andere +2+ overwegingen > hebben gehad
er zijn altijd < +2+ mensen > geweest die door < +2+ traditie > meer dan door < +2+ overtuiging > < +1+ hun +2+ stem > lieten bepalen die < +2+ traditie > is verzwakt
bovendien is < +1+ de principiele +2+ oordeelsvorming bij < +2+ velen > > verzwakt
< +1+ een +2+ verzwakking > die mede in < +1+ de +2+ hand > gewerkt wordt door < +2+ schrifturen als < +1+ het herderlijk +2+ schrijven van < +1+ de generale +2+ synode der < ned herv +2+ kerk > > > >

3.1.5. Sample output of the noun phrase isolation program

The program described in 3.1.4, based on the grammar of page 82, was applied to the text of the daily newspaper Trouw of June 19, 1956, as used in the word count [2]. This text was, of course, not employed during the construction of the grammar. As we did not want to specify the word classes (i.e. the non-terminals producing only terminal elements) in the program, we used a coded word list instead; for every word the classes it belonged to were given, not in the context, but to the word types in the frequency list. The 7000 word tokens from the newspaper text consisted of 2000 word types which were put in an array, together with their class codes. No sophisticated search procedure was used, although the listing according to frequency ensured that the most frequent words were on top of the list which was looked through.

A simple program first eliminated the punctuation signs from the text (except for the end-of-sentence signs) and replaced all numbers in digital form by the word XXgetal.

In the output on page 88 and 89 the beginning of two articles is reproduced with the isolated noun phrases put between angular brackets, the node *ante3* preceded by +1+, and the node *subsphrase* by +2+. For a discussion of the output we refer to 3.2 and 3.3, where the analyzed text is divided into a sequence of noun phrases (3.2) and a text with sentences in which the noun phrases are compressed into the highest nodes *ante3 subsphrase* (3.3).

The run time on the X8 to produce the output shown, was 160 seconds, 92 for the looking up of the words, and 68 for the analysis of the 26 sentences. Before the text is analyzed we need a half hour to store the word list.

3.2. Isolated noun phrases

A simple program delivers the noun phrases from a text analyzed as in 3.1.5. For the same articles as on the pages 88-89 the noun phrases are given on the pages 92-93. On page 94 a selection of interesting noun phrases from the same newspaper is given. In the output the nodes *ante3* and *subsphrase* highest in the derivation are preceded by +1 and +2, the lower ones as before. In isolating noun phrases we can make two kinds of errors: not isolating enough, or isolating too much. The first mistake is rare: almost all "intuitive" noun phrases are present in isolated noun phrases. Exceptions: een linde groene (title of an old song, with uncommon position of the adjective), the idiomatic de eerste de beste deur, and de problemen de gevangenen betreffend (with the participle after the substantive). More often too much was isolated, mainly for four reasons:

1. wrong choice of word class: see, e.g., the ambiguous word ons on page 93 which was also coded as substantive, which it is not here. In the same way weer and zeer give many erroneous noun phrases.
2. determiners can often play an independent role. Cf. the erroneous noun phrase het Chroestsjew on page 92.
3. adverbial modifiers are often interpreted as adjectival modifiers.
4. the sentence conjunction en is taken as a noun conjunction.

In our text we found an average of three noun phrases in a sentence; this number is very much determined by the style. In a noun phrase we found an average of three words. Noun phrases of one or two words account for more than half the number.

As regards structure: many niceties in the grammar were never encountered in the text! The greatest productivity has the rule allowing *prepphrase*s after a substantive. This even causes a bump at the word length 5 for noun phrases (de *S* van de *S*). A deeper analysis is only possible if we apply an improved grammar to a much longer text, after manually correcting the erroneous noun phrases (like we did in 1.2 with syllables). We hope to be able to do this in the future.

< italiaanse +2 kersen >
< +1 ons +2 artikel >
< +2 moskou >
< +2 belgrado >
< +2 rome >
< +1 een +2 vervolg >
< alle +2 artikelen in < +1+ deze +2+ rubriek > >
< +1 het +2 geval >
< +1 de +2 geschiedenis van < +2+ vandaag > >
< +1 genoemd +2 artikel >
< +1 een bijzonder +2 geval >
< +1 de derde +2 schakel van < +1+ de +2+ reeks > >
< +2 zeer >
< +1 het +2 vizier >
< +1 de recente +2 woorden van < +2+ togliatti > >
< +1 de +2 leider van < +1+ de italiaanse +2+ communisten > +2 palmiro togliatti>
< +1 zijn +2 vertrouwen in < +1+ het +2+ kremlin > >
< +2 moskou >
< +2 opheldering >
< +1 het +2 chroestsjew >
< +1 het +2 leven van < +2+ stalin > >
< +1 een zo monsterlijke +2 tiran >
< +1 het twintigste +2 congres te < +2+ moskou > >
< +2 stalin >
< +2 togliatti >
< ook +1 zijn +2 verdiensten voor < +1+ het +2+ communisme of +2+ socialisme>>
< +2 voorkeur >
< +2 stalin >
< +1 het door < +2+ moskou > geschilderde +2 beeld >
< +1 de huidige +2 kremlinheersers >
< +2 deugd >
< +1 een +2 middenkoers >
< +1 een onafhankelijke +2 koers van < +2+ moskou > >
< +1 een +2 vorm van < nationaal_communistische +2+ beweging >>
< +1 een +2 divergentie van < +1+ het +2+ kremlin als < +2+
dirigent van < +1+ het internationale +2+ communisme > > > >

< nadere +2 beschouwing >
< +1 de +2 uitslag van < +1+ de +2+ verkiezingen > >
< +1 een +2 zaak >
< +1 een +2 artikeltje >
< +2 weer >
< +1 de +2 orde van < +1+ de +2+ dag > >
< +1 een gevoelige +2 nederlaag >
< +2 zaak om < +2+ ons over < +1+ de +2+ aard > en < +1+ de +2+ oorzaken > > >
< +2 weer >
< +1 een +2 verkiezingsstrijd >
< vier +2 jaren >
< +1 de +2 aard van < +1+ de +2+ nederlaag > >
< +1 een goede +2 analyse van < +1+ de +2+ stemmenverschuiving > >
< +1 een gecompliceerd +2 geval >
< +2 sprake >
< +1 een opvallende +2 verschuiving >
< +1 de prot chr +2 partijen naar < +1+ de +2+ pvda > >
< +1 de +2 oorzaak >
< +2 verschuiving >
< +2 mensen >
< +1 de +2 doorbraak >
< theologische en politieke +2 gronden >
< +1 de +2 idee der < christelijke +2+ partijformatie > >
< +1 het +2 merendeel >
< andere +2 overwegingen >
< +2 mensen >
< +2 traditie >
< +2 overtuiging >
< +1 hun +2 stem >
< +2 traditie >
< +1 de principiele +2 oordeelsvorming bij < +2+ velen > >
< +1 een +2 verzwakking >
< +1 de +2 hand >
< +2 schrifturen als < +1+ het herderlijk +2+ schrijven van < +1+
de generale +2+ synode der < ned herv +2+ kerk > > > >

< +1 de bijzondere +2 positie van < +2+ italie als < specifiek
r-katholiek +2+ land > en als < zo belangrijke +2+
factor in < +1+ het +2+ middellandse-zeegebied > > > >
< op < +1+ het +2+ oog > zo grillige +2 gebaar van < +2+
chroestsjew op < +1+ een door < +1+ de britse +2+
ambassade > gegeven +2+ tuinfeest te < +2+ moskou > > > >
< +1 een +2 offensief ter < +2+ neutralisatie van < westelijke +2+ landen
> en ter < +2+ afbraak van < +1+ de +2+ navo > > > >
< +1 deze vanwege < +2+ stalin tot < +2+ stand > > gekomen +2 defensieformatie>
< +2 ambtenaren van < +1+ het centraal +2+ bureau voor < +1+ de
+2+ statistiek > en < +1+ het nationaal +2+ planbureau > > >
< ook +2 fabrieken en +2 ziekenhuizen en +2 scholen en +2 kerken >
< +1 een nieuwe +2 gestalte van < +1+ het +2+ kerk-zijn
in < +1+ een post-christelijke +2+ wereld > > >
< toch wel enige +2 suggesties >
< +1 de door < +1+ de geref +2+ bonders > bewaarde
+2 vruchten der < zgn nadere +2+ reformatie > >
< +1 het +2 streven naar < +2+ toenadering tot < bepaalde
+2+ groepen in < +1+ de ned herv +2+ kerk > > > >
< +2 voorzitter van < +1+ het internationaal +2+ comite ter < +2+
bestrijding van < +1+ het +2+ concentratiekampsysteem > > > >
< +1 de +2 kwestie van < +2+ schadevergoeding van < +2+ duitsland
aan < +1+ de nagelaten +2+ betrekkingen van < niet-duitse
expolitieke +2+ gevangenen > > > > >
< +2 verzwakking van < +1+ het +2+ elan in < +1+ de +2+ kerk > en
< +1+ de +2+ achteruitgang in < +1+ de +2+ politiek > > > >
< +2 drees als < in < menselijke +2+ zin > vertrouwenwekkende
+2+ figuur van < overwegende +2+ betekenis > > >
< ernstige zieke +2 plekken in < +1+ de geestelijke +2+ sector van
< +1+ de +2+ samenleving onzer < +2+ dagen > > > >

3.3. Sentences with compressed noun phrases

A simple program delivers the sentences in which the noun phrases are reduced to the determiner and central substantive from a text analyzed as in 3.1.5. For the same articles as on the pages 92-93, the compressed sentences are given on the pages 96-97. On page 98 another article has been compressed.

The compressed sentences remain almost always grammatical, even if they lose their meaning. Nongrammatical compressed sentences are mainly caused by two reasons:

1. wrong isolation of noun phrases. However, not every wrong noun phrase results in an ungrammatical sentence, e.g.the sentence with het Chroestsjew on page 96 is still grammatical.
2. subordinate clauses depending on a substantive which is not the central one. This is a consequence of the fact that we cannot recognize noun phrases containing finite verb forms.

After compression the text is reduced to 70% of the length.

As regards structure: if the first clause in the sentence has a subject (in an intuitive sense) this subject can be identified as the first isolated noun phrase not preceded by a preposition, if we now also admit pronouns in the nominative case as noun phrases. Counter-examples are easy to make, but were not encountered in the text. The finite verb form can also be characterized as standing before or after the subject, depending on the beginning of the sentence. It would be a good idea to isolate the prepositional phrases too, besides the noun phrases.

2 kersen
1 ons 2 artikel 2 moskou 2 belgrado 2 rome
behoeft 1 een 2 vervolg
zoals dit trouwens met 2 artikelen
1 het 2 geval
is want 1 de 2 geschiedenis zet zich morgen voort
maar met 1 genoemd 2 artikel
is het wel 1 een 2 geval
daar 1 de 2 schakel
moskou-belgrado-rome thans 2 zeer
duidelijk voor 1 het 2 vizier
komt met 1 de 2 woorden
1 de 2 leider 2 palmiro
heeft bekend dat 1 zijn 2 vertrouwen geschokt is
hij revolteert tegen 2 moskou
hij vraagt om 2 opheldering
hij rekent 1 het 2 chroestsjew
cs wel toe dat zij tijdens 1 het 2 leven
inactief zijn gebleven als deze 1 een 2 tiran
is geweest als deze op 1 het 2 congres is afgeschilderd
maar 2 stalin
was ook niet zo slecht zegt 2 togliatti
als hij nu afgeschilderd wordt hij had 1 zijn 2 verdiensten
zoals het thans bij 2 voorkeur wordt genoemd
men zal wat 2 stalin
betreft 1 het 2 beeld
dienen bij te schilderen en van 1 de 2 kremlinheersers
zal men wat 2 deugd
af moeten tellen en men zal zo op 1 een 2 middenkoers
uit moeten komen hetgeen dan uiteraard inhoudt 1 een 2 koers
1 een 2 vorm
1 een 2 divergentie

2 beschouwing
1 de 2 uitslag
is niet 1 een 2 zaak
waaraan men 1 een 2 artikeltje
wijdt om dan vervolgens maar 2 weer
spoedig tot 1 de 2 orde over te gaan
er is 1 een 2 nederlaag geleden
en het is dus nu 2 zaak
daarvan te beraden en te zien wat daaraan gedaan kan worden
want in XXgetal is er 2 weer
1 een 2 verkiezingsstrijd
en 2 jaren duren maar kort
1 de 2 aard
het is moeilijk om tot 1 een 2 analyse te komen
want dat is altijd 1 een 2 geval
maar het is duidelijk dat er om 2 sprake
is van 1 een 2 verschuiving
van stemmen van 1 de 2 partijen
1 de 2 oorzaak
van die 2 verschuiving is eveneens gecompliceerd
er zullen 2 mensen
bij betrokken zijn die principieel voor 1 de 2 doorbraak
hebben gekozen
die bewust op 2 gronden
1 de 2 idee hebben vaarwelgezegd
maar 1 het 2 merendeel
zal 2 overwegingen hebben gehad
er zijn altijd 2 mensen geweest die door 2 traditie
meer dan door 2 overtuiging
1 hun 2 stem lieten bepalen
die 2 traditie is verzwakt
bovendien is 1 de 2 oordeelsvorming verzwakt
1 een 2 verzwakking die mede in 1 de 2 hand
gewerkt wordt door 2 schrifturen

de 2 bouwnijverheid zal nog gedurende een lange 2 reeks 2 werkgelegenheid voldoende bieden want eerst 2 moet 1 het 2 woningbezit gebracht worden vervolgens moeten 2 krotten worden vervangen
voorts is er 1 de 2 industrialisatie en blijven er 2 bouwprojecten over die thans in 2 verband niet kunnen worden gebouwd
met 1 het 2 oog heeft 1 het 2 rijk destijds 1 de 2 woningbouw-touwtjes genomen 2 financieringsplan noemde 1 het 2 rijk 1 deze 2 touwtjes
2 bouwvolume zei 1 de 2 buitenwacht die 1 deze 2 term nogal eens misverstond we zijn door 1 ons 2 bouwvolume heen kon men tot voor kort van 2 wethouder horen
om te weten wat dit wilde zeggen moeten we even naar 1 de 2 burelen van de minister van 2 wederopbouw 2 volkshuisvesting
deze minister maakt 2 jaar 1 een 2 bouwprogramma 1 een 2 raming van wat in 1 het 2 begrotingsjaar alzo zal worden geinvesteerd in 1 de 2 bouwerij
daar wordt hij door heel wat 2 mensen bij geholpen oa door 2 ambtenaren
bij 1 de 2 samenstelling wordt van 1 allerlei 2 factoren uitgegaan
allereerst uiteraard van 1 de 2 bouwcapaciteit dus van 1 het 2 aantal 2 arbeiders 1 de 2 hoeveelheid 2 materialen het beschikbare particuliere en 2 overheidskapitaal etc 1 de 2 schatting 2 arbeiders is daarbij 1 het 2 moeilijkst want dit wisselt nogal oa door 2 seizoenwerkzaamheden als bv 1 de 2 kersenpluk die 1 de 2 arbeidsmarkt gedurende 1 een 2 periode ongunstig beinvloedt
men komt dan op 1 een 2 totaal 2 bedrag dat naar mag worden aangenomen zal worden geinvesteerd
voor XXgetal werd 1 dit 2 bedrag geraamd op XXgetal 2 gulden
op 2 basis wordt dan 1 het 2 bouwprogramma samengesteld en dat kan de minister van 2 wederopbouw 2 volkshuisvesting niet alleen doen want 1 zijn 2 ambtgenoten zitten aan die ruim 2 miljard te trekken omdat er nu eenmaal 2 fabrieken 2 ziekenhuizen 2 scholen 2 kerken moeten worden gebouwd
er komt dus 1 een 2 verdeling en daarbij 2 spelen 1 de 2 realisaties 1 een 2 rol

Sources Chapter 3

[1] H.F.A. van der Lubbe, Woordvolgorde in het Nederlands, Assen, 1958.

[2] The newspaper texts on microcards in:
J.A.Th.M. van Berckel a.o., Formal Properties of Newspaper Dutch, Mathematical Centre Tract 12, Amsterdam, 1965.

[3] Griffith and Petrick, On the relative efficiency of CF grammar recognizers, CACM, May 1965, p.289-300.

[4] C.H.A. Koster, Parsing and Translating, Mathematical Centre, Amsterdam 1970.

[5] Reino Kurki Suonio, On top-to-bottom recognition and left recursion, CACM, 9, 7, July 1966.

Chapter 4 SENTENCES AND THEIR MEANING: WORD PROBLEMS LEADING TO QUADRATIC EQUATIONS

4.1. Meaning in computational linguistics

The conviction is often expressed that computers cannot "understand" natural languages. It is, however, our conviction that in any situation where this "understanding" of natural language can be tested in an effective way, the computer is - at least in principle - capable of simulating the human understanding of natural language. A blind man can learn how to use the word "red" with all its connotations, although he will be unable to say correctly "this is red" in front of a red object. In much the same way, a computer can be programmed to handle natural language, but the way to test this capability can only be a verbal one, i.e., a computer can be taught to respond in natural language to a certain input in natural language, his responses resembling human ones (though not necessarily the way these responses are made) [1].
Several possibilities of giving an operational definition of "understanding" exist. Historically, the most important was the idea of mechanical translation of natural languages: a computer can be said to understand a certain text if it is capable of translating this text into another language. Theoretical and practical considerations show that this operational definition is - at least for the moment - too ambitious; by this strong definition, therefore, a computer is still incapable of "understanding" natural language texts. From a theoretical standpoint, many authors, of whom Bar-Hillel [2] was the most influential, have shown that for a high quality translation the computer must possess knowledge about the real world, which is unattainable at the moment; from a practical standpoint, ten years of heavy expenditures in mechanical-translation research have extinguished any hopes that "improved" word-by-word translations have a future [3]. The problem of mechanical translation has been replaced by the problem of finding the balance of work between human and machine intelligence. A weaker operational definition of computer "understanding"of natural language is a question-

answering system. Too ambitious information-retrieval systems fail for the same reason as mechanical-translation schemes. But in restricted areas, successful programs that answer questions about information sets have been made, as well for academic goals [4] as for practical goals [5].

A third possibility of operationally defining whether a computer has understood a text in a natural language, is present when we restrict ourselves to texts that order the reader to do something. The computer is then said to "understand" a certain text if it reacts in the same way a human reader would do. By this definition, of course, computers are understanding the texts of programs, which are always in the imperative mood, but these are not written in natural language. Natural language is unfit to write computer programs in [6], but certain restricted areas of natural languages can be used to teach a computer to understand texts. It would, e.g., be possible to connect a computer to a kitchen and have it prepare the dishes from a well-written cook book. We have taken the sections in Dutch high-school algebra text-books in which, after the treatment of quadratic equations, problems in natural language form are given which the students have to translate into algebraic language, and solve. We say that such a word problem is understood if the correct answer is found. The discourse analysis in this chapter is therefore limited both by its object (only text-book problems) and by its aim (only the finding of the correct solution). If in a problem the phrase "a rectangular garden" is used, the computer need not know what a garden is, except that it has length and breadth, and how to arrive at its circonference and surface.

The problem of solving an algebraic word problem can be divided into two parts:

1. Translation of the verbal text into algebraic text, in much the same way as, e.g., the algebraic propositions in book V of Euclid can be translated into algebraic formulae.
2. Solving of the algebraic equation. This last task is of course the easiest, and we are more interested in the first task.

A neat borderline between the linguistic and the mathematical task is not easy to find.

In section 4.2, we specify our aim, and we make a comparison with similar work of Bobrow [7,8].
In section 4.3, we give a description of the method we propose, by giving an example of how two word problems are treated.
In section 4.5 the program and its fixed input is reproduced. To make this program more readable, we explain the functions of identifiers and procedures in section 4.4.
The program in 4.5 is applied to a collection of problems, a.o. from three algebra books. The output of this is given in 4.6. Finally, in 4.7, we discuss the relevance of the program for other problems.

4.2. Our aim

Our aim was an ALGOL 60 program that would be capable of solving algebraic word problems leading to quadratic equations. From three Dutch high-school text-books [9,10,11] we selected a set of 22 problems which we considered representative. All the problems chosen have numbers only as objects. The program must be relevant in the sense that it could also solve other, similar, problems. We do not aim to simulate human behaviour: a schoolboy certainly does not work like our program, but we want to judge the program by comparing its results (i.e., the solutions found) with the human results.

The program tries to reduce all problems to a general equation form; we know we will not be able to solve all problems; however, we do not characterize the solvable problems. Certain mathematical limitations are given on page 112.

During program development we profited from insights of compiler construction, but by no means is text-book language seen as a programming language.

The program organization is simple: unlike the program of chapter 3 only one path is pursued; i.e. we do not return to an earlier decision point if a decision taken is found to be wrong.

Of similar work in this direction the nearest to ours is that of Bobrow [7,8] who has published a program called *student* that solves word problems leading to linear equations, and it would seem useful to compare our work with his. In favour of Bobrow is his view: "... I think that the *student* system could be made to understand most of the algebra story problems that appear in first year high-school text-books".

We are somewhat more sceptical about this possibility of expanding our program.

Against Bobrow we quote from his article:

"... though most algebra story problems found in the standard texts cannot be solved by *student* exactly as written, the author has usually been able to find some paraphrase of almost all such problems which is solvable by *student*". Our aim was to let the computer itself find the

paraphrase it can solve, in other words: we want to give as input the problems, exactly as written in the text-books. This necessitates a deeper linguistic analysis. While Bobrow can consider any sequence of nonspecial words as a noun phrase, we will consider every word in a problem as special, i.e., we will consider the relevant meaning of every word.

Another difference is, that we treated problems leading to quadratic equations, while Bobrow restricted himself to linear equations. This gave us the extra difficulties of handling possibly more than one equation (in possibly more than one unknown), one of which can be of degree 2, in order to unite them into a quadratic equation.

The last difference is, of course, that of the object language. We do not think, however, the difference between Dutch and English important for our application.

The main steps in achieving our aim are:

1. The translation of the problems from the text-book into a form using a minimal number of different words, each of which has a precise meaning. This translation is still readable for man.
2. The referencing. The same object (in our case: number) may be called by different names.
3. The syntactical analysis of this reduced problem. This means in practice: how to find the right operand(s) for each operator.
4. The determination of what exactly is asked - these are not always the roots of the quadratic equation, but, in general, a function of those roots. There may be an extra condition, e.g., only integer or positive solutions.
5. The choice of one unknown quantity in which the others are expressed and the construction of the quadratic equation.
6. The solution of the quadratic equation, and the deduction of the correct solution from its roots.

How we take these steps is described in the next section.

4.3. The solution method

As a first step a word problem is read from the punched tape, and coded. Three kinds of elements are distinguished: words, numbers and punctuation signs. Proper Dutch words, like kwadraten, consist of a string of letters. These words are searched in a vocabulary (see page 139) and represented by a code number. In this way, synonymous words like kwadraten and vierkanten are identified by their having the same code. Words that do not contribute to our understanding of the sentence (as defined operationally), like **de**, all receive the code empty. If, in a later stage, an intermediate output is given, then a prevailing synonym (e.g., kwadraten) is delivered even if another (e.g., vierkanten) was the original word. Although the code used has no intrinsic importance, it was set up in such a way that questions about classhoods could easily be formulated (table 15). In the program, all the codes are increased by one million in order to differentiate them from the second kind of elements in the problem text: integers, fractions, or integers with fractions, all represented as real numbers. No number in absolute value larger than 1 000 000 can be expressed.

Punctuation signs, the end-of-problem symbol, etcetera are coded just as normal Dutch words; their code is related in a simple way to the flexowriter code that we happen to use.

The expression x-voud (multiple) contains all three kinds of elements and is stored as three numbers: the number x, the code for the punctuation sign -, and the code for the operator voud.

After the problem is read from the punched tape and coded, it consists of a string of numbers, some of which, below one million, are representations of numbers (i.e., they denote themselves), whereas the others denote words or punctuation signs. By successive steps, this string has to be brought into the form of a quadratic equation. We illustrate these steps as they are applied to two of our problems. The exact formulation, of course, can only be found in the ALGOL-program in section 4.5 of which identifiers and procedures are explained in 4.4.

TABLE 15 : coding system

all codes for words are enlarged by one million.

numbervalue		unary operator		bin/tern. operator		tern. operator	
x1	1	omgekeerde	11	verschil	21	som3	30
x2	2	tweedemacht	12	quotiënt	22	produkt3	31
x3	3	derdemacht	13	middelevenredige	23		
eerste	4	tweemaal	14				
andere	5	drievoud	15				
evenveel	6						
getal	7	(n-)voud	17				
Last unknown	8			som	28		
				produkt	29		

plurals are coded by adding 50 :

getallen	57	tweedemachten	62
Last unknowns	58	derdemachten	63

empty has code 301 ; new word has code 100.

variables in transformation: *g1* 401 *gw1* 404 *opr* 407
g2 402 *gw2* 405
g3 403 *gw3* 406

punctuation signs are coded by adding 500 to the flexowriter code.
All other words receive codes between 300 and 400.

The two problems are given on the punched tape as:

(A) Telt men bij een getal het omgekeerde van dat getal op, dan krijgt men 5 25/48. Welk getal wordt bedoeld?

and

(B) Zoek drie opeenvolgende getallen, waarvan het produkt 21 maal zo groot is als de som.

The problems are searched for certain keywords that determine what is asked: one, two or three unknowns. Also additional constraints on these solutions like:only positive or integer,are extracted (for simplicity, these additional conditions, if valid, are always required of all solutions of a problem). The operators som and produkt may have two or three operands and indications about this number of operands are searched. Words that have to do with the phrasing of the question are made empty and thus disappear. Our two problems now look as follows:

(A1) telt men bij een getal het omgekeerde van dat getal, dan krijgt men +5.52083.

Question: one number. No constraints.

(B1) drie opeenvolgende getallen, waarvan het produkt + 21.00000 maal zo groot is als som.

Question: three numbers, produkt and som may need three operands.

Now, synonymous strings of words are identified. To this end, a list of substitutions has been read in by the program (see page 125). Each of these consists of a left-hand side and a right-hand side. Whenever the words in a left-hand side occur as a string in the problem text, they are to be replaced by the words in the right-hand side. Some elements of a left-hand side are not Dutch words but variables running over numbers or operators. For instance, in our two problems, paraphrases of the operations som and produkt occur that are handled by the substitutions:

(T1) , dan krijgt men => is

(T2) telt men bij => som van

(T3) *g1* maal zo groot is als => is produkt van *g1*

Here, the word *g1* (and later in the same fashion *g2* and *g3*) stands for any real number. When (T3) is applied to (B1) then *g1* gets the value 21 and this value has to be substituted for *g1* in the right-hand side. Similarly, *gw1*, *gw2* and *gw3* standing for numbers or words that can play the part of numbers and *opr* standing for operators,occur in the transformations. Just as in the case of the vocabulary, this list of transformations is kept independent of the ALGOL-program which makes it easier to make amendments. Our ideal is that for each new problem it may be necessary to expand the vocabulary and the transformation list but not the program. Apart from (T1), (T2) and (T3), the following substitutions are of interest for our problems (A) and (B):

(T4) het *gw1* => *gw1*

(T5) een getal => *x1*

(T6) opeenvolgende => verschil 1 hebbende

(T7) drie verschil *g1* hebbende => *x1 x2 x3* is (verschil (*x2*,*x1*),*g1*) is (verschil *(x3*,*x2)*,*g1)*.

gw1 in (T4) stands for number values; thus it eliminates het in front of omgekeerde in (A1) and in front of produkt in (B1). Substitution (T6) is an example of a simple word expanded into a string: "consecutive" is translated into "having difference 1". It is clear that transformation (T7) has to be applied after the application of transformation (T6). The order of the transformations is essential information. The transformations (T6) and (T7) together deliver two complete equations of first degree that are stored after the problem text and protected against further analysis in the linguistic part of the program. When all substitutions are made, our problems will become:

(A2) som van *x1* omgekeerde van dat getal is + 5.52083.

(B2) *x1 x2 x3* waarvan produkt is produkt van + 21.00000 som. is (verschil *(x3, x2)*, + 1.00000) is (verschil (*x2, x1*), + 1.00000)

One and the same object (in our problems the objects are always numbers) can be denoted in several ways. Sometimes, the vocabulary already tells us that two different words refer to the same number. But certain referential words oblige us to find out what they refer to. This is done by references, i.e., transformations of the same appearance as the substitutions. In the first of our two examples the following reference is applicable:

(T8) dat getal => *last unknown*
where *last unknown* refers to the first unknown to the left of dat getal.
As there is only one unknown in (A2), this reference results in:

(A3) som van *x1* omgekeerde van *x1* is + 5.52083.

Problem (B2) is not affected by referential transformations.
Substitutions and references are listed on page 140 and page 141.
After these transformations the main linguistic problem is tackled: for each operator its operand(s) must be found. From right to left, the operators in the problem (as the deeper-embedded operators will usually be more to the right, we want to find their operands first, so that they, in their turn, can act as operands of other operators; no recursive search process was necessary)search their operands as follows:

1. If the word van occurs just to the right of the operator (or, if the operator is followed by en and another operator, then if van occurs after that second operator), then the operands are expected to follow the word van. Situation:
   ... de som van twee getallen ... .

Otherwise

2. From the operator in question towards the left, an occurence of waarvan is searched without crossing a punctuation sign. If such a waarvan is found, then the operands are expected to be just in front of it. If waarvan is preceded by en, then the operands are expected in front of a previous waarvan. Situation:
   ... twee getallen, waarvan de som ... .

Otherwise

3. If the search for waarvan is stopped by a punctuation sign or at

the first word in the problem, a search for van is begun from that stopping point towards the right, towards the operator. If a van is encountered, the operands will be expected to follow it.
Situation:
... van twee getallen is de som ... .

When the place of the operand(s) is determined in this way, then it is, of course, necessary to know the form of that (those) operand(s) (a number, an unknown, or an operator with its operands) and to know how many operands the operator needs. When operands have been found, they may not be spoiled in such a way that they can no longer play the part of operands for other operators as well. E.g., in (B2) the first occurrence of the operator produkt and the operator som, both take the three numbers *x1*, *x2* and *x3* as their operands. When operands are found, they are copied, properly bracketed and placed behind their governing operators. When the operators have, in this way, received their operands, then remaining candidates for operands can be destroyed.

Special care is necessary for operators in the plural form, like kwadraten which have to be transformed into coordinations of operators in the singular form.

Repeated application of this operand-finding process to problem (A3) first finds the operand of omgekeerde (unary operator) and yields

(A4) som van *x1* omgekeerde *(x1)* is + 5.52083.

Then it finds the two operands of som (in this case a binary operator) and yields

(A5) som *(x1,* omgekeerde *(x1))* is + 5.52083.

In problem (B2), the search for operands will not enter the complete equations delivered by (T7). Repeated application of the operand-finding process for the three operators in (B2), ultimately gives

(B3) waarvan produkt3 *(x1, x2, x3)* is produkt (+ 21.00000, som3 *(x1, x2, x3)*).

Note that if produkt has three operands it is denoted by produkt3.

At this point the words voud en evenveel require special attention, the first to be rewritten as a produkt,the second to find its counterpart als.

We are now in a position to formulate the equations to which our problem leads. From left to right, at any occurrence of waarvan, one tries to construct the left-hand side and the right-hand side of an equation. Then, again from left to right,at occurrences of is or number values, one tries to find equations. All other words are destroyed. The results of this step on our two problems are

(A6) is(som *(x1,* omgekeerde *(x1))*, + 5.52083)

and

(B4) is (verschil *(x3, x2)*, + 1.00000) is (verschil *(x2, x1)*, + 1.00000)
is (produkt3 *(x1, x2, x3)*, produkt (+ 21.00000, som3 *(x1, x2, x3)*)).

This terminates the linguistic part of our problem. The remaining part is purely mathematical: the several equations, limited for practical reasons to 3, in several unknowns (at most three) have to be united into one equation with one unknown, with formulae to express the other unknowns in the solution of the equation. This is done by investigating each of the equations found in the problem; as soon as an equation is found to be of degree higher than one,or to have more than two unknowns, then it is taken as the main equation. The other equations (which are expected to be linear and to contain at most two unknowns) serve to make one or two unknowns explicit, i.e., to express them into terms of the other unknown, which will be the main unknown.

These expressions are then substituted into the main equation. In most of the problems we encountered in algebra text-books,there is one linear equation between two unknowns (e.g., their sum or difference is known) and one equation of the 2^{nd}degree in these two unknowns (e.g., their product is known). In that case, one unknown is expressed into the other one by the linear equation, then substituted into the second equation, which is subsequently solved. In our example (A6), we have only one

equation with one unknown, which makes this step an easy one. In example (B4), however, we must not only recognize that the last equation is to be the main equation, but also that *x2* is to be the main unknown. The unknowns *x1* and *x3* are expressed in *x2* and substituted, which gives:

(B5) Is (produkt3 (verschil (*x2*,1), *x2*, som (*x2*,1)), produkt (21,
som3 (verschil (*x2*,1), *x2*, som (*x2*,1)))).
x1 = verschil (*x2*,1).
x3 = som (*x2*,1).

Now that the equations (A6) and (B5) are found, we have essentially ALGOL statements which can be executed when the procedures produkt, som, omgekeerde etcetera are defined in an appropriate way. To this end the primary program described so far, punches a secondary program, defining the solution of a quadratic equation in terms of procedures produkt, som, omgekeerde, etcetera, after which the primary program punches the ALGOL-statements (A7) and (B6), and a standard ending. As the secondary program is output of the primary program, the primary program can be said to solve the word problem completely automatically.
In the secondary program the following information is required for each problem:

1. The main equation given as a procedure statement.
2. The way in which the numbers asked for (at most three) depend on the roots of the main equation.
3. The two Boolean conditions: *Integer* and *Positive* that can be imposed on the solutions.

In our two problems, the calls of the procedure *solve* in the secondary program will be:

(A7) **solve(is (som (x,omgekeerde (x)), +5.52083),x, 0, 0, false , false);**
(B6) **solve(is (produkt3 (som(verschil(0,+1.00000) ,produkt(+1.00000 ,x)),
x, som(+1.00000 ,produkt(+1.00000 ,x))),produkt (+21.00000 ,som3
(som(verschil(0,+1.00000) ,produkt(+1.00000 ,x)),x, som(+1.00000 ,
produkt(+1.00000 ,x))))), som(verschil(0,+1.00000) ,produkt(+1.00000
,x)),x, som(+1.00000 ,produkt(+1.00000 ,x)), false , false);**

These quadratic equations are then solved by the formula $x_{1,2} = (-b \pm \sqrt{(b^2 - 4ac)})/(2a)$. The equation (B6) is of course of the third degree, but the case where the third root of the equation is zero can be solved. Equation (A7) needs a multiplication by x to become a proper quadratic equation. The secondary program delivers at most three sets of solutions.

In our case these solutions were:

(A8) 5.3333
.1875

(B7) 7, 8, 9
-9, -8, -7
-1, 0, 1

We have reached our goal: the correct solutions of the equations concealed in the texts (A) and (B).

4.4. Identifiers and procedures explained

Following the order of the program in 4.5, we explain the functions of identifiers and procedures.

As global variables (on page 121) with variable integer value are declared: *length problem*, viz., the number of words in each problem, determined in the input procedure *read problem; length vocabulary, number of transformations, number of substitutions* and *number of references* which speak for themselves; *number of unknowns* which is determined and used in the procedure *treatment of operators*, and *symbol*, which is given a value in the input procedure *read symbol*, and which saves the value of the last symbol read on the input tape.

The remaining integers are constants. If they are Dutch words or English names for punctuation or layout symbols, then their value is the code on page 107 , whereas *G* is the number to differentiate real numbers from code values; it is given the value one million.

The Boolean variables *Positive* and *Integer* are used to store additional conditions on the solutions of the quadratic equation; their value is transferred as the last two parameters of the procedure *solve*.

The integer array *V* stores the words in the vocabulary (up to 150), each word in (at most five) groups of four letters; *C* contains the code value of the corresponding words in *V* and *transformation* stores the substitutions and references (together at most 70). The first index of *transformation* gives the number of the transformation; the second index is 1 for the left-hand side, and 2 for the right-hand side of the transformation; the third index points to the (at most 40) code values of the words in the left- or right-hand side of the transformation. In this way *transformation* [*i*, 1, *k*] stores the code of the k'th word in the left-hand side of the i'th transformation. Some of the words in *transformation* may be variables standing for number values or operators. Corresponding to the first index in *transformation*, the array *trf increase* stores the effect on the length of the problem string by application of a transformation; it is filled in the procedure *read transformations*. The integer array *position operand* is used in the procedures *put operand* and *treatment of operators*. The integer array

Word is used for temporary storage of the 5 groups of 4 letters of a word under consideration.
There are two real arrays (they have to store real numbers occurring in the problems as well as code values): *P* is the problem string in which the original problem is read and in which the transformation into the final equation is effected, the array *variable in trf* is used to store the actual value of a number or operator which occurs in the left-hand side of a transformation.
After these global identifiers, a number of procedures is declared which can be divided into: input procedures, output procedures, Boolean procedures to facilitate questions about the classhood of words, string procedures operating on *P*, and the six remaining procedures of more intrinsic interest.

The input procedures (pages 121, 122, 123, 125):
read symbol gives *symbol* a value, according to the code to be found in [12], and prints the symbol it has read. As we always work with the code for words given in table 15, it is not necessary to know the code for symbols. This procedure is used in all of the following input procedures. The procedure *omit layout* reads symbols until *symbol* has a value different from space, tabulator stop or new line. The procedure *read real* is called as soon as *symbol* has the value of a digit. It will read numbers of any of the three forms: 3, 3 7/22, or 7/22; its value will be the real representation of such a (fractional) number. The procedure *read word* is called as soon as *symbol* has the value of a letter (the other symbols in a word may be digits). It will read the word in groups of four letters each (at most 5 groups) in the array *Word*. This procedure is used in: *read vocabulary* that fills the arrays *V* and *C*, *assign code* that reads a word and delivers the value of its code by consulting *V* and *C*, *read problem* that fills *P*, and *read transformations* that fills *transformation* and *trf increase*.

The output procedures (on page 124):
print word and *punch word* print or punch the word of which the code is given as a parameter (thus the sequence *read word; print word* gives as

output the prevailing synonym of the word in the input). They both use the procedure *out* in which the second parameter is a Boolean telling whether the printer or the puncher is to be used. The procedure *print present P* gives us the state of the problem string.

The Boolean procedures (on page 123) are used to formulate questions about symbols and words without worrying about the codes used. They are: *digit (symbol), letter (symbol), singular operator (word), plural operator (word), unary operator (word), operator (word), number value (word), number variable (word), unknown (word)* and *interpunction (word).* The Boolean procedure *complete equation (start index)* answers the question whether, at position *start index* in *P*, a complete equation starts.

The "string procedures" all work on the problem string *P* (on page 126). They are:
shift (from, over), garbage, omit between brackets, destroy unit, length of complete unit, put copy at, put at, put operand. (the last one on page 129).
The effect of *shift (from, over)* is: if *over* has a positive value, then the problem string is lengthened by an amount *over*, which is inserted before the element with index *from*; it is used if we want to insert strings into the problem string. If *over* has a negative value, then the effect is a shortening of the problem string by the amount *over*; in this way words in *P* are destroyed.
The procedure *garbage* destroys the places in *P* where meaningless words were stored.
The procedure *omit between brackets (index, direction)* gives the place after (or before) *index* (depending on *direction*) while omitting strings between brackets; it is often used in searching the problem string without taking into account strings that were put between brackets earlier. Such a bracketed unit is measured by the procedure *length of complete unit*, replaced by the procedure *put (i) at (j)*, copied (i.e., it also remains in its old place) by *put copy (i) at (j)*, and destroyed by *destroy unit. Put operand* manipulates operands with the aid of *put copy.*

After these auxiliary procedures, six procedures of general interest are discussed, in the order in which the main program on page 138 will call them for each of the problems:

1. The procedure *determine question* (on page 125) gives values to the Booleans *Positive* and *Integer* and to *number of unknowns* by searching for certain clue words in the problem text. Superfluous words used to phrase the question in the problem are eliminated.
2. The procedure *apply substitutions* (on page 127) uses the first part of the transformation list to substitute certain word strings by others. It does so by calling, for each of the substitution transformations, the procedure *here starts substitution (i)* which looks whether the left-hand side of that substitution can be found in the problem string. If so, it is replaced by the corresponding right-hand side. Special care is taken to transfer number values or operators for which a variable in the transformation is used. If complete equations are found, then they are placed at the far right side of *P*.
3. The procedure *apply references* (on page 128) uses the second part of the transformation list to find where certain referential expressions refer to. It does so by calling, for each of the words in *P*, the procedure *try reference (j)* that looks whether the left-hand side of a referential transformation starts at that word. If so, it is replaced by the corresponding right-hand side. Again, variables for numbers or operators are filled with the appropriate values. Also, the array *last unknowns* is filled and used here.
4. The procedure *treatment of operators* (pages 129, 130, 131) runs from right to left through the problem text, and for each operator it encounters, it searches for its operands. The strategy for finding the operands was explained in 4.3. Special attention is demanded by the words voud and evenveel. The first requires a number in front of it, the second requires the word als followed by a number value. When the position of the operands is found, then the procedure *take operands (j, direction)* will identify the operands to the left (or right, depending on *direction*) of the place with index *j*, and will

give values to the array *position*. If necessary, the order of the operands is changed (verschil and quotient). At last, operands that can no longer be used are destroyed.

5. The procedure *make equations* (on page 132) calls at certain clue words (like is) the procedure *make equation from (j)* which in its turn calls the procedure *take operands of equation (j)* identifying the two sides of an equation. Finally one, two or three equations, properly bracketed, are made in *P*.
6. The procedure *make call of solve* (pages 133,134,135) must be able to construct the quadratic equations from these, at most three, equations.

   To this end, the array *M* is used to see in which equations, which of the unknowns occur. The real array *C* saves the coefficients of the equations. The simpler equations are identified by simulating the operators until an equation of a degree higher than the first is encountered. This complicated equation is termed the main equation. From the other equation(s), an unknown is made explicit by a call of *explicit (x) from: (Eq)*. The word middelevenredige requires special measures.

   Finally, the following output is punched:

   the string 'solve(';
   the main equation, with the unknowns all in terms of the main unknown;
   the three unknowns (by the procedure *punch unknown*);
   the values of the Boolean *Integer* and *Positive*;
   the string ');'.

   Thus, the call of *solve* is constructed and punched, which can be applied by the secondary program. This concludes the discussion of the six intrinsic procedures in the program.

The **secondary** program (on pages 136,137) simply solves the quadratic equation with the formula $x_{1,2} = (-b \pm \sqrt{(b^2 - 4ac)})/(2a)$ and prints the values of the three unknowns (possibly three sets of solutions for each problem).

In the main program (on page 138), first the integers with constant value are given their value, and the beginning of the secondary program is

punched. Then vocabulary, transformations and *number of problems* are read in. For each problem the six procedures explained above are called. Whenever the problem text is such that the program is unable to continue with the solution, an empty equation is delivered to the secondary program, and the next problem is tackled.

```
begin comment word problems leading to quadratic equations;
integer length problem, length vocabulary, number of transformations,
number of substitutions, number of references, number of unknowns,
j, number of problems, problem number, symbol,

x1, x2, x3, left bracket, right bracket, becomes, end symbol,
solve, new word, space, tab, new line,
G, x, Last unknown, Last unknowns, g1, g2, g3, gw1, gw2, gw3, opr,
comma, empty, period, question word, semicolon, stroke, hyphen,

als, drie, en, evenveel, geheel, getallen, hebbende, is, is
middelevenredige, middelevenredige, produkt, produkt3, quotient,
som, som3, twee, van, verschil, volgende, voud, waarvan;

Boolean Positive, Integer;

integer array V[1:150,1:5], C[1:150], transformation[1:70,1:2,1:40],
trf increase[1:70], position operand[1:3], Word[1:5];

real array P[1:200], variable in trf[1:7];

procedure read symbol;
begin symbol:= RESYM; PRSYM(symbol)
end read symbol;

procedure omit layout;
begin integer i;
   i:= 0; for i:= i + 1 while symbol = space ∨ symbol = tab ∨
   symbol = new line do read symbol
end omit layout ;

real procedure read real;
begin integer simple number, numerator, denominator;

   integer procedure number part;
   begin integer i, h;
      i:= h:= 0;
      for i:= i + 1 while digit(symbol) do
      begin h:= h × 10 + symbol; read symbol end;
      omit layout; number part:= h
   end number part is read ;

   numerator:= 0; denominator:= 1; simple number:= number part;
   if symbol = stroke then
   begin numerator:= simple number; simple number:= 0 end
   else numerator:= number part;
   if symbol ≠ stroke then goto real constructed else read symbol;
   denominator:= number part;
real constructed: read real:= simple number+numerator/denominator
end read real;
```

```
    procedure read word;
    begin integer i, j;
       for i:= 1 step 1 until 5 do
       begin Word[i]:= 0; if letter(symbol) ∨ digit(symbol) then
          for j:= 1, 36, 1296, 46656 do
          if letter(symbol) ∨ digit(symbol) then
          begin if symbol > 35 then symbol:= symbol - 27;
            Word[i]:= Word[i] + j × symbol; read symbol
          end comprimation of four lower case letters
       end 20 letters read ;
       i:= 0;
       for i:= i + 1 while letter(symbol) ∨ digit(symbol) do
       read symbol
    end read word;


    procedure read vocabulary;
    begin integer i, j;
       length vocabulary:= read;
       for i:= 1 step 1 until length vocabulary do
       begin j:= 0;
          for j:= j + 1 while ¬letter(symbol) do read symbol; read word;
          C[i]:= read;
          for j:= 1 step 1 until 5 do V[i,j]:= Word[j];
          ABSFIXT(3, 0, C[i]); NLCR
       end word and its code read
    end read vocabulary;


    real procedure assign code;
    if letter(symbol) then
    begin integer i, j;

       integer procedure vocabulary index;
       begin for i:= 1 step 1 until length vocabulary do
       if Word[1] = V[i,1] then
          begin if Word[2] = V[i,2] ∧ Word[3] = V[i,3] ∧
          Word[4] = V[i,4] ∧ Word[5] = V[i,5] then
            begin vocabulary index:= i;
             goto vocabulary index found
            end all letters agree
          end first four letters agree;
          vocabulary index:= new word - G;
       vocabulary index found:
       end vocabulary index;

       read word; j:= vocabulary index; assign code:= C[j] + G
end
else if digit(symbol) then assign code:= read real else
begin assign code:= symbol + G + 500; read symbol end;
```

```
procedure read problem;
begin integer i;
   i:= 0;
   for i:= i + 1 while ¬(letter(symbol) ∨ digit(symbol)) do read
   symbol; i:= 0;
word of problem: i:= i + 1; omit layout;
   if i = 200 then
   begin too long: P[200]:= assign code; if P[200] =
      end symbol then goto next problem else goto too long
   end problem too long;
   P[i]:= assign code; if P[i] = end symbol then
   length problem:= i else goto word of problem
end read problem ;

Boolean procedure digit(symbol); integer symbol;
digit:= symbol ≥ 0 ∧ symbol ≤ 9;

Boolean procedure letter(symbol); integer symbol;
letter:= symbol ≥ 10 ∧ symbol ≤ 63;

Boolean procedure interpunction(word); real word;
interpunction:= word = period ∨ word = semicolon ∨ word = comma;

Boolean procedure singular operator(word); real word;
singular operator:= word ≥ G + 10 ∧ word < G + 50;

Boolean procedure plural operator(word); real word;
plural operator:= singular operator(word − 50);

Boolean procedure unary operator(word); real word;
unary operator:= word ≥ G + 10 ∧ word ≤ G + 19;

Boolean procedure operator(word); real word;
operator:= singular operator(word) ∨ plural operator(word);

Boolean procedure number value(word); real word;
number value:= word < G + 10 ∨ singular operator(word);

Boolean procedure unknown(word); real word;
unknown:= word = G + 1 ∨ word = G + 2 ∨ word = G + 3;

Boolean procedure complete equation(start index);
integer start index;
complete equation:= if start index ≥ length problem
then false else (P[start index] = is ∧ P[start index + 1] = left bracket);

Boolean procedure number variable(word); value word; real word;
number variable:= word = g1 ∨ word = g2 ∨ word = g3 ∨ word = gw1 ∨
word = gw2 ∨ word = gw3;
```

```
procedure E(alarm message); string alarm message;
begin NLCR; PRINTTEXT(alarm message); goto next problem end;


procedure out(code, punch); value code; real code; Boolean punch;
begin

   procedure out number(a); value a; real a;
   if punch then FIXP(6, 5, a) else FIXT(6, 5, a);

   procedure out symbol(symbol); value symbol; integer symbol;
   if punch then PUSYM(symbol) else PRSYM(symbol);

   if code < G then out number(code) else if code > G + 500 then out
   symbol(code - G - 500) else
   begin integer i, j, m, letter;

      procedure decode;
      begin integer i, j;
           for j:= 1 step 1 until length vocabulary do
         if C[j] = code - G then
           begin for i:= 1 step 1 until 5 do Word[i]:= V[j,i];
              goto decoded
           end;
      decoded:
      end decode;

      decode;
      for i:= 1 step 1 until 5 do if Word[i] > 0 then
      for j:= 1, j + 1 while Word[i] > 0 do
      begin m:= Word[i] : 36; letter:= Word[i] - (m × 36);
           Word[i]:= m; out symbol(letter)
      end;
      out symbol(space)
   end output of word
end out;

procedure print word(code); value code; real code;
out(code, false);

procedure punch word(code); value code; real code;
out(code, true);


procedure print present P;
begin integer i;
   NLCR;
   for i:= 1 step 1 until length problem do print word(P[i]);
   NLCR
end print present P;
```

```
procedure read transformations;
begin integer i, j, lr, code;
   number of substitutions:= read; number of references:= read;
   number of transformations:= number of substitutions + number of
   references;
   for i:= 1 step 1 until number of transformations do
   begin ABSFIXT(2,0,i); j:= 0; read symbol;
      for j:= j + 1 while ¬letter(symbol) ∧ ¬interpunction(symbol + G
      + 500) do read symbol; lr:= 1; trf increase[i]:= 0;
   read other side: j:= 0;
      for j:= j + 1 while j ≤ 40 do
      begin omit layout;
            trf increase[i]:= trf increase[i] + (if lr=1 then -1 else 1);
            code:= assign code;
            transformation[i,lr,j]:= code; if code = becomes then
            begin read symbol; lr:= 2; goto read other side end
            else if code = end symbol then goto transformation read;
            if code = empty then
            begin
            trf increase[i]:= trf increase[i] - (if lr=1 then -1 else 1);
            j:= j - 1
            end empty code
      end;
   transformation read:
   end
end read transformations;


procedure determine question;
begin integer i;
   number of unknowns:= 1; Positive:= Integer:= false;
   for i:= 1 step 1 until length problem do
   begin if P[i] = twee then number of unknowns:= 2 else if P[i] =
      drie then number of unknowns:= 3 else if P[i] = getallen ∧
      number of unknowns = 1 then number of unknowns:= 2 else if
      P[i] = geheel then
      begin Integer:= true; P[i]:= empty end
      else if P[i] = volgende then Positive:= true
   end;
   for i:= length problem - 2 step - 1 until 2 do
   begin if operator(P[i]) then goto ready;
      if P[i] = question word then
      begin if interpunction(P[i - 1]) then
            begin length problem:= i; P[i]:= end symbol end
      end question word;
      if interpunction(P[i]) then goto ready
   end;
ready: for i:= 1 step 1 until length problem do if P[i] = question
   word ∨ P[i] = new word then P[i]:= empty
end determine question;
```

```
procedure shift(from, over); value from, over; integer from, over;
if over ≠ 0 then
begin integer first, last, n, i;
   n:= sign(over); first:= if n = 1 then length problem else from-over;
   last:= if n = 1 then from else length problem;
   for i:= first step - n until last do
   begin P[i + over]:= P[i]; P[i]:= empty end;
   length problem:= length problem + over;
   for i:= 1, 2, 3 do if position operand[i] > from then position
   operand[i]:= position operand[i] + over; if n = 1 then
   for i:= first + 1 step 1 until from + over - 1 do P[i]:= empty
end shift;
procedure garbage;
begin integer i;
   for i:= length problem step - 1 until 1 do
   if P[i] = empty then shift(i, - 1)
end garbage;
integer procedure length of complete unit(start index);
integer start index;
length of complete unit:= if P[start index + 1] = left bracket then
omit between brackets(start index + 1, true) - start index else 1;
integer procedure omit between brackets(index, direction); value index;
integer index; Boolean direction;
begin integer n;
   n:= if direction then 1 else - 1;
   if index < 1 then omit between brackets:= 1 else if index > length
   problem then omit between brackets:= length problem else if
   direction∧P[index]≠left bracket ∨ ¬direction∧P[index]≠right bracket
   then omit between brackets:= index + n else
   begin integer close, i;
      close:= if direction then right bracket else left bracket;
      i:= index + n;
      for i:= omit between brackets(i, direction) while i ≥ 1 ∧ i ≤
      length problem do if P[i] = close then
      begin omit between brackets:= i + n;
         goto between brackets omitted
      end;
   between brackets omitted:
   end
end omit between brackets;
procedure destroy unit(start index); integer start index;
shift(start index, - length of completeunit(start index));
procedure put copy of(i)at: (j); integer i, j;
begin integer k, l;
   k:= length of completeunit(i); shift(j, k);
   if i > j then i:= i + k;
   for l:= i step 1 until i + k - 1 do P[j + l - i]:= P[l]
end put copy of i at j;
procedure put(i)at: (j); value i, j; integer i, j;
begin put copy of(i)at: (j); destroy unit(i) end;
```

```
Boolean procedure agree(Pword, trf word); value Pword, trf word;
real Pword, trf word;
if Pword = trf word then agree:= true else if number
variable(trf word) then
begin if Pword < G ∨ (number value(Pword) ∧ trfword > gw1) then
   begin variable in trf[trf word - G - 400]:= Pword; agree:= true
   end
   else agree:= false
end
else if trf word = opr then
begin if operator(Pword) then
   begin variable in trf[trf word - G - 400]:= Pword; agree:= true
   end
   else agree:= false
end
else agree:= false;


procedure apply substitutions;
begin integer i, j, k;

   integer procedure here starts substitution(i); integer i;
   begin integer j, k, l;
      for j:= 1 step 1 until length problem - 1 do
      begin k:= -1; l:= 0;
            for k:= k + 1 while agree(P[j + k], transformation[i,1,k+1])
            do l:= k + 1;
            if transformation[i,1,l + 1] = becomes then
            begin here starts substitution:= j; goto substitution found
            end
      end;
      here starts substitution:= 0; substitution found:
   end here starts substitution;

   garbage; PRINTTEXT(‡applied substitutions:‡);
   for i:= 1 step 1 until number of substitutions do
   begin
   L: j:= here starts substitution(i); if j > 0 then
      begin shift(j, trf increase[i]); k:= 0;
            for k:= k + 1 while transformation[i,2,k] ≠ end symbol do
            P[j+ k - 1]:= if number variable(transformation[i,2,k]) then
            variable in trf[transformation[i,2,k] - G - 400] else
            transformation[i,2,k]; ABSFIXT(2,0,i); goto L
      end
   end transformation i searched
end apply substitutions;
```

```
procedure apply references;
begin integer i, j, k, last unknown, number of unknowns;
   integer array last unknowns[1:3];
   procedure try reference(j)at: (i); integer j, i;
   begin integer l, m, n; l:= - 1; m:= 0;
      for l:= l + 1 while agree(P[i + l], transformation[j,1,l + 1])
      do m:= l + 1; if transformation[j,1,m + 1] = becomes then
      begin shift(i, trf increase[j]); m:= 0;
         for m:= m + 1 while transformation[j,2,m] ≠ end symbol do
         begin if transformation[j,2,m] = Last unknowns then
            begin if number of unknowns = 0 then
               begin print(j); E(‹impossible reference›) end
               else
               begin shift(i + m - 1, number of unknowns - 1);
                  for n:= 1 step 1 until number of unknowns do
                  P[i + k + n - 2]:= last unknowns[n];
                  k:= number of unknowns - 1
               end
            end last unknowns
            else if transformation[j,2,m] = Last unknown then
            begin if last unknown = empty then
               begin ABSFIXT(2,0,j); E(‹impossible reference›) end
               else P[i + k - 1]:= last unknown
            end last unknown
            else if transformation[j,2,m] = opr then P[i + k - 1]:=
            variable in trf[opr - G - 400] else if number variable(
            transformation[j,2,m]) then P[i + k - 1]:=variable in trf
            [transformation[j,2,m] - G - 400] else P[i + k - 1]:=
            transformation[j,2,m]; k:= k + 1
         end;
         ABSFIXT(2,0,j)
      end reference found
   end try reference;

   PRINTTEXT(‹applied references:›); i:= number of unknowns:= 0;
   last unknown:= empty; k:= 1;
   for i:= i + k while i ≤ length problem ∧ ¬complete equation(i) do
   begin k:= 1;
      for j:= number of substitutions + 1 step 1 until number of
      transformations do try reference(j)at: (i); if unknown(P[i]) then
      begin if unknown(P[i + 1]) then
         begin last unknowns[1]:= P[i]; last unknowns[2]:= P[i + 1];
            if unknown(P[i + 2]) then
            begin number of unknowns:= k:= 3;
               last unknowns[3]:= P[i + 2]
            end
            else number of unknowns:= k:= 2
         end last unknowns
         else last unknown:= P[i]
      end unknowns;
      end
end apply references;
```

```
procedure put operand(m)at: (k); value m; integer m, k;
begin integer j;
   j:= position operand[m]; put copy of(j)at: (k);
   if P[k + 1] = left bracket then k:= omit between brackets(k + 1,
   true) - 1
end put operand m at k;


procedure treatment of operators;
begin integer k, m, i, space increase, number of operands, possible
   number of operands;
   Boolean useful operands, plural;

   procedure take operands(j, direction); value j; integer j;
   Boolean direction;
   begin integer m;
      integer array position[0:3];
      number of operands:= 0;
      for j:= omit between brackets(j, direction) while j ≥ 1 ∧ j <
      length problem do if number value(P[j]) then
      begin number of operands:= number of operands + 1;
            position[number of operands]:= j;
            if number of operands ≥ possible number of operands then
            goto L
      end
      else ifP[j] = en ∨ P[j] = comma then P[j]:= empty else if
      ¬(P[j] = left bracket ∧ direction ∨ P[j] = right bracket ∧
      ¬direction ∨ P[j] = empty) then goto L;
   L: if number of operands = 0 then goto M;
      if position[1] > position[number of operands] then
      begin
            m:= position[1];
            position[1]:= position[number of operands];
            position[number of operands]:= m
      end interchange operands;
      for m:= 1, 2, 3 do
      begin if position operand[m] > position[number of operands]
            then destroy unit(position operand[m]);
            position operand[m]:= if m ≤ number of operands then
            position[m] else 0
      end;
   M: if useful operands then goto operands found else goto unused
      operands
   end take operands;
```

```
procedure search operands(op); integer op;
begin integer m, l, Pop;
   Pop:= P[op]; if ¬plural then
   begin possible number of operands:= if unary operator(Pop) then
         1 else if Pop ≠ som ∧ Pop ≠ produkt then 2 else
         if number of unknowns = 3 then 3 else 2
   end
   else possible number of operands:= if number of unknowns = 3
   then 3 else 2; useful operands:= true;

VAN to right: m:= op; l:= m + 1;
   for m:= m + 1 while P[m] = en ∨ operator(P[m]) do l:= m + 1;
   if P[l] = van then take operands(l, true);
   m:= op; if m < 3 then E(‡no operands for leftmost operator‡);
WAARVAN: for m:= omit between brackets(m, false) while m ≥ 2 ∧
   ¬interpunction(P[m]) do if P[m] = waarvan then
   begin if P[m-1]=en then
      begin m:=m-1; goto WAARVAN end; take operands(m, false)
   end else l:= m - 1;

VAN to left:
   for l:= l, omit between brackets(l, true) while l < op-1 do
   if P[l] = van then take operands(l, true)
end search operands;

garbage; m:= i:= length problem - 1; number of operands:=0;
for m:= omit between brackets(m, false) while complete equation(m)
∨ interpunction(P[m]) do i:= m;
for i:= omit between brackets(i, false) while i > 1 do
begin if operator(P[i]) ∧ P[i + 1] ≠ left bracket then
   begin plural:= plural operator(P[i]); search operands(i);
   operands found: if number of operands ≠ possible number of
        operands then
        begin if ¬(possible number of operands = 3 ∧ number of
           operands = 2) then E(
           ‡incorrect number of operands‡)
        end;
        k:= if P[i + 1] = van then i + 1 else i;
        space increase:= if plural then - 1 else 1;
        for m:= 1 step 1 until number of operands do
        space increase:= space increase + (if plural then 3 else 1);
        shift(k + 1, space increase); if plural then
        begin P[i]:= P[i] - 50;
           for m:= 1 step 1 until number of operands do
           begin if m ≠ 1 then P[k]:= P[i]; P[k + 1]:= left bracket;
              k:= k + 2; put operand(m)at: (k);
              P[k + 1]:= right bracket; k:= k + 2
           end
        end plural
        else
```

```
            begin P[k + 1]:= left bracket; k:= k + 2;
               for m:= 1 step 1 until number of operands do
               begin put operand(m)at: (k); P[k + 1]:= comma; k:= k + 2
               end; P[k - 1]:= right bracket
            end singular operator;
            if P[i + 1] = van then
            begin for m:= 1 step 1 until number of operands do
              begin destroy unit(position operand[m]);
               position operand[m]:= 0
            end; shift(i + 1, - 1);
            end;print present P
      end operators;
      if P[i] = voud then
      begin m:=i; for m:=omit between brackets(m,false) while m>2 do
          begin if P[m]=hyphen then
            begin if P[m-1]>G then E(‹ no number before hyphen›) else
              begin shift(i+2,2);P[i]:=produkt;P[i+2]:=P[m-1];P[i+3]:=
             comma;P[m]:=P[m-1]:=empty; garbage; goto voud found
          end end end; voud found: print present P
      end voud;
      if P[i] = evenveel then
      begin m:= i;
            for m:= omit between brackets(m, true) while m ≤ length
            problem do if P[m] = als then
            begin shift(m, if P[m + 1] = is then - 2 else - 1);
               if number value(P[m]) then
               begin destroy unit(i); put(m - 1)at: (i);
                 goto evenveel value found
            end end;
            E(‹no evenveel value found›);
       evenveel value found: print present P
   end end;
   for m:= 1 step 1 until number of operands do if position
   operand[m] > 0 then
   begin destroyunit(position operand[m]); position operand[m]:= 0
   end;
   for i:= 1 step 1 until length problem do if interpunction(P[i]) ∨
   P[i] = waarvan then shift(1, - i + 1) else if ¬unknown(P[i])
   then goto L;
L: i:= 0; possible number of operands:= 3;
   useful operands:= false;
   for i:= omit between brackets(i, true) while i ≤ length problem
   do if P[i] = van ∨ P[i] = waarvan then
   begin if P[i] = van then
      begin P[i]:= empty; take operands(i, true) end
      else take operands(i, false);
   unused operands: for m:= 1 step 1 until number of operands do
      if unknown(P[position operand[m]]) then P[position
      operand[m]]:= empty
   end;   garbage
end treatment of operators;
```

```
procedure make equations;
begin integer i, l;

   procedure make equation from(j); value j; integer j;
   begin integer l, n;

      procedure take operands of equation(j); value j; integer j;
      begin n:= 0;
            for j:= omit between brackets(j, true) while j < length
            problem ∧ ¬complete equation(j) ∧ P[j]≠period ∧ P[j]≠semicolon do
         if number value(P[j]) then
            begin if n = 2 then goto eq found; n:= n + 1;position
               operand[n]:= j
            end
            else if P[j] = is then
            begin P[j]:= empty; if n = 2 then goto eqfound end
            else if P[j] ≠ left bracket ∧ P[j] ≠ empty then goto eqfound;
      eqfound: if n ≠ 2 then
         begin ABSFIXT(3,0,j); E(≤ operand of is missing≥) end;
         if P[position operand[1]] < G then
         begin
            n:= position operand[1];
            position operand[1]:= position operand[2];
            position operand[2]:= n
         end put number as last operand
      end take operands of equation ;

      take operands of equation(j); n:= length problem; shift(n, 4);
      P[n]:= is; P[n + 1]:= left bracket; n:= n + 2;
      for l:= 1, 2 do
      begin put operand(l)at: (n); P[n + 1]:= comma; n:= n + 2 end;
      P[n - 1]:= right bracket;
      for l:= 1, 2 do destroyunit(position operand[l])
   end make equation from j;

   i:= 0;
   for i:= omit between brackets(i, true) while i ≤ length problem ∧
   ¬complete equation(i) do
   if P[i] = waarvan then
   begin make equation from(i); shift(i, - 1) end;
   i:= 0;
   for i:= omit between brackets(i, true) while i ≤ length problem ∧
   ¬complete equation(i) do
   if ¬(P[i] = is ∨ number value(P[i])) then
   P[i]:= empty else make equation from(i - 1); garbage
end make equations;
```

```
procedure make call of solve;
begin integer i, j, v, X, main, eq, index, k;
   integer array M[1:3,1:5];
   real array C[G + 1:G + 100,1:4];

   procedure makezero;
   begin for i:= 1, 2, 3, 4 do
      for j:= 1 step 1 until 100 do C[G + j,i]:= 0;
      for i:= 1, 2, 3, index, eq do
      for j:= 1, 2, 3 do M[j,i]:= 0;
   end make zero;

   procedure make(a, bi); value a; integer a; real bi;
   for i:= 1, 2, 3, 4 do C[a,i]:= bi;
   integer procedure next;
   begin v:= v + 1; if v > G + 100 then E(
      <problem too complicated>); next:= v
   end next;

   integer procedure Som(a, b); value a, b; integer a, b;
   if a = G ∨ b = G then Som:= G else
   begin Som:= next; make(v, C[a,i] + C[b,i])
   end Som;
   integer procedure Verschil(a, b); value a, b; integer a, b;
   if a = G ∨ b = G then Verschil:= G else
   begin Verschil:= next; make(v, C[a,i] - C[b,i])
   end Verschil;
   Boolean procedure Constant(c); value c; integer c;
   if c = G then Constant:= false else Constant:= C[c,1] = 0 ∧
   C[c,2] = 0 ∧ C[c,3] = 0;
   integer procedure Produkt(a, b); value a, b; integer a, b;
   if a = G ∨ b = G ∨ ¬Constant(a) ∧ ¬Constant(b) then Produkt:= G
   else
   begin Produkt:= next;
      if Constant(a) then make(v, C[a,4] × C[b,i]) else
      make(v, C[b,4] × C[a,i])
   end Produkt;
   integer procedure Quotient(a, b); value a, b; integer a, b;
   if a = G ∨ ¬Constant(b) then Quotient:= G else
   begin Quotient:= next; make(v, C[a,i] / C[b,4])
   end Quotient;

   integer procedure explicit(x)from: (Eq); value x, Eq; integer x, Eq;
   if ¬unknown(x+G) ∨ Eq = G ∨ C[Eq,x] = 0 then explicit:= G else
   begin integer j;
      explicit:= next;
      make(v, if i = x then 0 else - C[Eq,i] / C[Eq,x]); j:= 0;
```

```
    for j:= j + 1 while j ≤ 3 ∧ X = 0 do if j ≠ x ∧ C[Eq,j] ≠ 0
    then
    begin X:= j + G; NLCR; print word(X); PRINTTEXT(‹is x›) end
 end explicit;
 integer procedure I(index); value index; integer index;
 if index = 0 then I:= G else
 begin integer op1, op2, op3, K; real w;
    w:= P[index]; op1:= I(index operand(index, 1));
    op2:= I(index operand(index, 2));
    op3:= I(index operand(index, 3));
    if w = is ∨ w = verschil then K:= Verschil(op1, op2) else if w
    = som ∨ w = som3 then K:= Som(op1, if w = som then op2 else
    Som(op2, op3)) else if w = produkt ∨ w = produkt3 then K:=
    Produkt(op1, if w = produkt then op2 else Produkt(op2, op3))
    else if w = quotient then K:= Quotient(op1, op2) else if
    unknown(w) then K:= w else if w < G then
    begin K:= next; make(v, if i = 4 then w else 0) end
    else K:= G; I:= K
 end I;

 integer procedure index operand(index operator, n);
 value index operator, n; integer index operator, n;
 if P[index operator + 1] = left bracket then
 begin integer k, l;
    if n = 1 then
    begin index operand:= index operator + 2; goto K end;
    l:= index operator + 2; k:= 1;
    for l:= omit between brackets(l, true) while P[l] ≠ right
    bracket ∧ l < length problem do if P[l] = comma then
    begin k:= k + 1; if k = n then
          begin index operand:= l + 1; goto K end
    end;
    index operand:= 0;
K:
 end
 else index operand:= 0;
 procedure punch Boolean(Boolean); Boolean Boolean;
 if Boolean then PUTEXT(‹ true ›) else PUTEXT(‹ false ›);
 procedure punch unknown(a); integer a;
 if a = X then punch word(x) else
 begin integer A;
    A:= M[main,a-G]; if A = 0 then
    begin PUTEXT(‹ 0›); goto K end;
    if C[A,X - G] < 0 then PUTEXT(‹ verschil( ›) else PUTEXT(
    ‹ som( ›); if C[A,4] < 0 then PUTEXT(‹ verschil(0,›);
    punch word(abs(C[A,4]));
    if C[A,4] < 0 then punch word(right bracket); PUTEXT(
    ‹ ,produkt( ›); punch word(abs(C[A,X - G])); PUTEXT(‹ ,x)›);
K:
 end punch unknown;
```

```
    k:= 0; v:= G + 4; index:= 4; eq:= 5; make zero;
    for j:= 1, 2, 3 do make(G + j, if i = j then 1 else 0);
    for j:= 1 step 1 until length problem do
    begin if P[j] = som ∨ P[j] = produkt then
       begin if index operand(j, 3) > 0 then P[j]:= P[j] + 2
       end operator3;
       if P[j] = is then
       begin k:= k + 1; if k > 3 then E(‡too many  equations‡);
             M[k,index]:= j; i:= index operand(j, 2);
             if P[i] = middelevenredige then
             begin put(i - 1)at: (j + 2); put(i)at: (j + 2)
             end middelevenredige put first;
             if P[j + 2] = middelevenredige then
             begin shift(index operand(j, 2) - 2, - 1); shift(j, - 2);
                P[j]:= is middelevenredige
             end is middelevenredige
       end is
       else if unknown(P[j]) then M[k,P[j] - G]:= 1
    end;
    main:= X:= 0;
    for k:= 1, 2, 3 do
    begin if M[k,index] ≠ 0 then M[k,eq]:= I(M[k,index]);
       if M[k,eq] = G ∨ (M[k,1] ≠ 0 ∧ M[k,2] ≠ 0 ∧ M[k,3] ≠ 0) then
       begin if main = 0 then main:= k else E(
             ‡more than one complicated equation‡)
       end main equation found
    end equations;
    if main = 0 then main:= 1;
    if SUM(j,1,3,sign(SUM(k,1,3,M[k,j])))>SUM(k,1,3,sign(
    M[k,index])) then E(‡more unknowns then equations‡);
    for j:= 3, 2, 1 do if M[main,j] ≠ 0 ∧ X ≠ j + G then
    begin for k:= 1, 2, 3 do if M[k,j] ≠ 0 ∧ k ≠ main then
       begin M[main,j]:= explicit(j)from: (M[k,eq]);
             goto made explicit
       end;
    made explicit:
    end;
    if X = 0 then X:= x1; j:= 0; i:= M[main,index]; PUTEXT(
    ‡:  solve(‡); punch word(P[i]); i:= i + 1;
    for i:= i, i + 1 while j ≠ 0 do
    begin if P[i] = left bracket then j:= j + 1 else if P[i] = right
       bracket then j:= j - 1;
       if unknown(P[i]) then punch unknown(P[i]) else punch word(P[i])
    end output main equation ;
    for j:= x1, x2, x3 do
    begin punch word(comma); punch unknown(j)
    end output unknowns;
    punch word(comma); punch Boolean(Integer); punch word(comma);
    punch Boolean(Positive); PUTEXT(‡);
‡);
end make call of solve;


PUTEXT(‡
```

```
begin integer G;
G:= 1000000;
begin integer i, j, x, v, problem number;
    array C[G + 1:G + 300, - 3:3];
    procedure solve(eq, s1, s2, s3, Integer, Positive);
    value eq, s1, s2, s3; integer eq, s1, s2, s3;
    boolean Positive, Integer;
    begin real a, b, c, D, x1, x2, xx, s;
        boolean x3;
        procedure E(s); string s;
        begin PRINTTEXT(s); NLCR; goto EE end;
        ABSFIXT(4, 0, problem number);
        problem number:= problem number + 1; NLCR; if eq = 0 then E(
        ⟨no  equation  found⟩); x3:= false;
    COEFF: a:= C[eq,2]; b:= C[eq,1]; c:= C[eq,0];
        if C[eq,3] ≠ 0 then
        begin for i:= 0 step - 1 until - 3 do if C[eq,i] ≠ 0 then E(
            ⟨degree more than 2⟩); c:= b; b:= a; a:= C[eq,3]; x3:= true
        end;
        if C[eq, - 1] ≠ 0 ∨ C[eq, - 2] ≠ 0 ∨ C[eq, - 3] ≠ 0 then
        begin for i:= 3 step - 1 until - 2 do C[eq,i]:= C[eq,i - 1];
            C[eq, - 3]:= 0; goto COEFF
        end;
        if a = 0 then
        begin if b = 0 then E(⟨a=b=0⟩); x1:= x2:= - c / b end
        else
        begin D:= b × b - 4 × a × c; if D < 0 then E(
            ⟨solutions complex⟩); D:= sqrt(D);
            x1:= ( - b + D) / (2 × a); x2:= ( - b - D) / (2 × a)
        end;
        for xx:= x1, x2, 0 while x3 do
        begin if x3 ∧ xx = 0 then x3:= false;
            for s:= s1, s2, s3 do
            begin if s ≠ 0 then
                begin s:= SUM(i, - 3, 3, C[s,i] × (xx ↑ i));
                    if (Integer ⌉ (s - entier(s) < 5₁₀ - 5)) ∧ (Positive
                    ⌉ s > 0) then FIXT(4, 4, s)
                end
            end;
            NLCR
        end;
    EE: NLCR; initialize
    end;
    procedure initialize;
    begin for i:= - 3 step 1 until 3 do
        for j:= 1 step 1 until 300 do C[G + j,i]:= 0; v:= x:= G + 1;
        make(x, if i = 1 then 1 else 0)
    end;
    procedure make(a, bi); value a; integer a; real bi;
    for i:= - 3 step 1 until 3 do C[a,i]:= bi;
```

```
integer procedure next;
begin v:= v + 1; if v > G + 300 then EXIT; next:= v
end next;

integer procedure g(a); value a; real a; if a < G then
begin g:= next; make(v, if i = 0 then a else 0) end
else g:= a;

integer procedure som(a, b); value a, b; real a, b;
begin integer A, B;
   A:= g(a); B:= g(b); som:= next; make(v, C[A,i] + C[B,i]);
end som;
integer procedure verschil(a, b); value a, b; real a, b;
begin integer A, B;
   A:= g(a); B:= g(b); verschil:= next; make(v, C[A,i] - C[B,i])
end verschil;
integer procedure produkt(a, b); value a, b; real a, b;
begin integer A, B;
   A:= g(a); B:= g(b); produkt:= next;
   make(v, SUM(j, - 3, 3, C[A,j] × (if i - j < 4 ∧ i - j > - 4
   then C[B,i - j] else 0)));
end produkt;
integer procedure tweedemacht(a); value a; real a;
tweedemacht:= produkt(a, a);
integer procedure derdemacht(a); value a; real a;
derdemacht:= produkt(tweedemacht(a), a);
integer procedure omgekeerde(a); value a; real a;
begin integer A;
   A:= g(a); omgekeerde:= next;
   make(v, if i = 0 then (if C[A,0] ≠ 0 then 1 / C[A,0] else 0)
   else C[A, - i])
end omgekeerde;
integer procedure quotient(a, b); value a, b; real a, b;
quotient:= produkt(a, omgekeerde(b));
integer procedure is middelevenredige(a, b, c); value a, b, c;
real a, b, c;
is middelevenredige:= verschil(produkt(a, b), tweedemacht(c));
integer procedure som3(a, b, c); value a, b, c; real a, b, c;
som3:= som(som(a, b), c);
integer procedure produkt3(a, b, c); value a, b, c; real a, b, c;
produkt3:= produkt(produkt(a, b), c);
integer procedure is(a, b); value a, b; real a, b;
is:= verschil(a, b);

problem number:= 1; initialize;
```

```
});  RUNOUT;
G:=1000000;x1:=G+1;x2:=G+2;x3:=G+3;left bracket:=G+598;
right bracket:=G+599;becomes:=G+570;end symbol:=G+627;x:=G+533;
space:=93;tab:=118;new line:=119;stroke:=67;
Last unknown:=G+8;Last unknowns:=G+58;
g1:=G+401;g2:=G+402;g3:=G+403;gw1:=G+404;gw2:=G+405;gw3:=G+406;opr:=G+407;
comma:=G+587;period:=G+588;question word:=G+622;semicolon:=G+591;
hyphen:=G+565;evenveel:=G+6;getallen:=G+57;middelevenredige:=G+23;
produkt:=G+29;produkt3:=G+31;quotient:=G+22;som:=G+28;som3:=G+30;
verschil:=G+21;voud:=G+17;new word:=G+100;j:=300;
begin integer  procedure N;begin N:=G+j;j:=j+1 end;
   solve:=N;empty:=N;als:=N;drie:=N;
   en:=N;geheel:=N;hebbende:=N;is:=N;is middelevenredige:=N;
   twee:=N;van:=N;volgende:=N;waarvan:=N
end give values to constants;


read vocabulary;NEWPAGE;
read transformations;NEWPAGE;
number of problems:=read ;
for  problem number :=1 step 1 until number of problems do
begin ABSFIXP(3,0,problem number);
   ABSFIXT(3,0,problem number);NLCR;
   for j:=1,2,3 do position operand[j]:=0;
   read  problem ;print present P;
   determine question;garbage;print present P;
   PRINTTEXT(‹asked:›); ABSFIXT(1,0,number of unknowns);
   PRINTTEXT(‹numbers›);
   if Positive then PRINTTEXT(‹,Positive›);
   if Integer  then PRINTTEXT(‹,Integer›);NLCR;
   apply substitutions; j:=length problem ;
   for j:=omit between  brackets(j,false) while j>1 do
   if complete  equation (j) then put(j)at:(length problem);
   print present P;apply references; print present P;
   treatment of  operators;print present P;
   make  equations;print present P;
   make call of solve;
   goto problem solved;
next  problem: PUTEXT(‹:  solve(0,0,0,0,false,false);
});
problem solved: NEWPAGE;
end ;PUTEXT(‹
end
end
›)
end
```

107

x1	1	g1
x2	2	g2
x3	3	g3
eerste	4	gw1
oorspronkelijke	4	gw2
kleinste	4	gw3
zelf	4	opr
andere	5	solve
grootste	5	empty
nieuwe	5	als
evenveel	6	drie
getal	7	en
getallen	57	de
omgekeerde	11	gelijk
tweedemacht	12	aan
kwadraat	12	op
factoren	57	geheel
delen	57	gehele
lastunknown	8	hebbende
lastunknowns	58	is
tweedemachten	62	bedraagt
kwadraten	62	wordt
vierkanten	62	ismiddelevenredige
derdemacht	13	twee
derdemachten	63	van
tweemaal	14	der
drievoud	15	uit
voud	17	volgende
verschil	21	waarvan
quotient	22	zodat
middelevenredige	23	zijn
som	28	het
produkt	29	verkregen
som3	30	een
produkt3	31	bij
		dan

401	elkaar	319
402	groot	320
403	groter	326
404	hebben	322
405	in	323
406	krijgt	324
407	maal	325
300	meer	326
301	men	327
302	met	328
303	minder	329
304	ontbind	330
301	opklimmen	331
348	telt	332
301	tot	333
301	verdeel	334
305	verhouden	335
305	vermindert	336
306	zich	337
307	zo	301
307	zodanig	301
307	dier	340
308	hun	341
309	opeenvolgende	342
310	opvolgende	342
310	verschillen	344
310	die	345
311	bedoeld	622
312	newword	100
312	bepaal	622
313	bereken	622
314	welke	622
315	welk	622
316	zoek	622
317	dat	346
318	samen	347

```
                    41      11
 1  het gw1          =>      gw1 |
 2  dier             =>      van die |
 3  opeenvolgende    => verschil 1 hebbende |
 4  getal g1         =>      g1 |
 5  het een          =>      eerste |
 6  verdeel g1 in twee getallen => x1 x2 is(som(x1,x2),g1) |
 7  drievoud         =>      3-voud |
 8  gw1 meer is dan  =>      is som van gw1 |
 9  gw1 minder is dan gw2    =>      is verschil van gw2 en gw1 |
10  ontbind g1 in twee factoren => x1 x2 is(produkt(x1,x2),g1) |
11  twee verschil g1 hebbende getallen       =>
            x1 x2 is(verschil(x2,x1),g1) |
12  drie getallen die g1 verschillen         =>
            drie verschil g1 hebbende getallen |
13  drie verschil g1 hebbende getallen       =>
            x1 x2 x3 is(verschil(x2,x1),g1) is(verschil(x3,x2),g1) |
14  twee op elkaar volgende getallen         =>
            x1 x2 is(verschil(x2,x1),1) |
15  drie op elkaar volgende getallen         =>
            x1 x2 x3 is(verschil(x2,x1),1) is(verschil(x3,x2),1) |
16  zijn samen       =>      waarvan som is |
17  tweemaal         =>      2 maal |
18  g1 maal zo groot als     =>      produkt van g1 |
19  g1 maal zo groot is als  =>      is produkt van g1 |
20  hebben g1 tot som        =>      waarvan som is g1 |
21  g1 : g2  =>      g1 en g2 |
22  g1 : g2 : g3     =>      g1, g2 en g3 |
23  twee getallen verhouden zich als g1 en g2        =>
            x1 x2 is(produkt(x2,g1),produkt(x1,g2)) |
24  drie getallen verhouden zich als g1, g2 en g3    =>
    x1 x2 x3 is(produkt(x2,g1),produkt(x1,g2)) is(produkt(x3,g2),produkt(x2,g3))
25  drie getallen die met g1 opklimmen       =>
            x1 x2 x3 is(verschil(x2,x1),g1) is(verschil(x3,x2),g1) |
26  een getal                =>      x1 |
27  twee getallen            =>      x1 x2 |
28  drie getallen            =>      x1 x2 x3 |
29  die g1 verschillen       =>      waarvan verschil is g1 |
30  verschillen g1           =>      waarvan verschil is g1 |
31  ,dan krijgt men          =>      is |
32  telt men bij     op      =>      som van |
33  g1 van                   =>      produkt van g1 |
34  eerste getal             =>      x1 |
35  andere getal             =>      x2 |
36  zodanig , dat            =>      waarvan |
37  g1 maal                  =>      produkt van g1|
38  gw1 minder dan gw2       => verschil van gw2 en gw1 |
39  gw1 meer dan             =>      som van gw1 |
40  is gw1 gelijk aan        => gw1 is |
41  gelijk   => |
```

```
42  eerste                  =>      x1 |
43  andere                  =>      x2 |
44  getal                   =>      x1 |
45  hun opr                 =>      opr van lastunknowns |
46  zijn opr                =>      opr van lastunknown |
47  dat getal               =>      lastunknown |
48  die getallen            =>      lastunknowns |
49  getallen                =>      lastunknowns |
50  zijn g1-voud            =>      g1-voud van lastunknown |
51  dat                     =>      |
52  die                     =>      |
```

4.6. Solutions of word problems

On the next 18 pages the application of the program and its fixed input is shown in 18 problems. The first two were used as examples in 4.3. The next fourteen show interesting points of our method. The last four were made by a teacher, who did not know our program, and were given to his class (for the result see 4.7).
For each problem is given:

In flexowriter fount: the part of the input tape stating the word problem.
In normal fount: the intermediate results as delivered on the lineprinter, showing a.o. which transformations were applied (the number refers to the list on pages 140, 141), how the operands were found, and finally the one, two or three equations and the unknown chosen as the main unknown.
Of three (long) problems not all the intermediate output has been reproduced.
In flexowriter fount: the punched output which is a procedure call in the secondary program.
In normal fount: the results of the secondary program as delivered to the lineprinter.
And finally a comment, in italics.
The runtime on the X8 was three minutes, one minute for the ALGOL translation, and two for the solving of the 18 problems.

Telt men bij een getal het omgekeerde van dat getal op,dan krijgt men 5 25/48. Welk getal wordt bedoeld? |

```
telt men bij een getal het omgekeerde van dat getal empty,
dan krijgt men  + 5.52083 .? getal is ??|
telt men bij een getal het omgekeerde van dat getal,
dan krijgt men  + 5.52083 .|
asked: 1 numbers
applied substitutions:  1  26  31  32
som van x1 omgekeerde van dat getal is  + 5.52083 .|
applied references: 47
som van x1 omgekeerde van x1 is  + 5.52083 .|
som van x1 omgekeerde (x1 ) is  + 5.52083 .|
som (x1 , omgekeerde (x1 )) is  + 5.52083 .|
som (x1 , omgekeerde (x1 )) is  + 5.52083 .|
is (som (x1 , omgekeerde (x1 )),  + 5.52083 )|
```

solve(is (som (x,omgekeerde (x)), +5.52083),x, 0, 0, false , false);

```
+ 5.333
  +.1875
```

This was example (A) in 4.4.

Zoek drie opeenvolgende getallen, waarvan het produkt 21 maal zo groot is als de som. |

```
? drie opeenvolgende getallen, waarvan het produkt  + 21.00000
maal empty groot is als empty som .|
drie opeenvolgende getallen, waarvan het produkt  + 21.00000
maal groot is als som .|
asked: 3 numbers
applied substitutions:  1  3  13  19
x1 x2 x3, waarvan produkt is produkt van  + 21.00000 som.
is (verschil (x3 , x2 ),  + 1.00000 ) is (verschil (x2 , x1 ), + 1.00000 )|
applied references:
x1 x2 x3, waarvan produkt is produkt van  + 21.00000 som.
is (verschil (x3 , x2 ),  + 1.00000 ) is (verschil (x2 , x1 ), + 1.00000 )|
is (produkt (x1 , x2 , x3 ), produkt (  + 21.00000 , som (x1 , x2 , x3 ))
)|
x2 is x
```

```
solve(is (produkt3 ( som(  verschil(0,      +1.00000 ) ,produkt(       +1.00000
,x)),x, som(       +1.00000  ,produkt(       +1.00000  ,x))),produkt (
    +21.00000 ,som3 ( som(  verschil(0,      +1.00000 ) ,produkt(       +1.00000
,x)),x, som(       +1.00000  ,produkt(       +1.00000  ,x))))), som(  verschil(0,
     +1.00000 ) ,produkt(       +1.00000  ,x)),x, som(       +1.00000  ,produkt(
     +1.00000  ,x)), false , false );
```

```
+ 7.0000    + 8.0000    + 9.0000
- 9.0000    - 8.0000    - 7.0000
- 1.0000      +.0000    + 1.0000
```

This was example (B) in 4.4.

Het verschil van twee getallen is 7 en het produkt is 228; bepaal die getallen. |

```
het verschil van twee getallen is  + 7.00000 en het produkt is  + 228.00000
;? die getallen .|
het verschil van twee getallen is  + 7.00000 en het produkt is  + 228.00000;|
asked: 2 numbers
applied substitutions:  1  1  27
verschil van x1 x2 is  + 7.00000 en produkt is  + 228.00000 ;|
applied references:
verschil van x1 x2 is  + 7.00000 en produkt is  + 228.00000 ;|
verschil van x1 x2 is  + 7.00000 en produkt (x1 , x2 ) is  + 228.00000 ;|
verschil (x1 , x2 ) is + 7.00000 en produkt (x1 , x2 ) is  + 228.00000 ;|
verschil (x1 , x2 ) is + 7.00000 en produkt (x1 , x2 ) is  + 228.00000 ;|
is (verschil (x1 , x2 ), + 7.00000 ) is (produkt (x1 , x2 ), + 228.00000 )|
x1 is x
```

```
solve(is (produkt (x, som( verschil(0,     +7.00000 ) ,produkt(      +1.00000  ,
x))),  +228.00000 ),x, som( verschil(0,     +7.00000 ) ,produkt(
+1.00000  ,x)), 0, false , false );
```

```
+ 19.0000   + 12.0000
- 12.0000   - 19.0000
```

Notice how produkt *finds the right operands.*

Ontbind 1080 in twee factoren,waarvan de som 74 is. |

```
ontbind + 1080.00000 in twee getallen, waarvan empty som + 74.00000 is.|
ontbind + 1080.00000 in twee getallen, waarvan som + 74.00000 is.|
asked: 2 numbers
applied substitutions: 10
x1 x2 , waarvan som + 74.00000 is. is (produkt (x1 , x2 ), + 1080.00000 )|
applied references:
x1 x2 , waarvan som + 74.00000 is. is (produkt (x1 , x2 ), + 1080.00000 )|
x1 x2 empty waarvan som (x1 , x2 ) + 74.00000 is . is (produkt (x1 , x2 ),
+ 1080.00000 )|
waarvan som (x1 , x2) + 74.00000 is . is (produkt (x1 , x2 ), + 1080.00000 )|
is (produkt (x1 , x2) + 1080.00000 ) is (som (x1 , x2 ), + 74.00000 )|
x1 is x
```

**solve(is (produkt (x, verschil(+74.00000 ,produkt(+1.00000 ,x))),
+1080.00000),x, verschil(+74.00000 ,produkt(
+1.00000 ,x)), 0, false , false);**

```
+ 20.0000    + 54.0000
+ 54.0000    + 20.0000
```

Notice how som *finds the right operands.*

Zoek twee opeenvolgende getallen,zodat het verschil van de derdemacht en de tweedemacht van het grootste getal gelijk is aan het 25-voud van het kleinste. |

```
?twee opeenvolgende getallen, waarvan het verschil van empty derdemacht
en empty tweedemacht van het andere getal gelijk is  empty het
  + 25.00000 -voud van het eerste .|
twee opeenvolgende getallen, waarvan het verschil van derdemacht
en tweedemacht van het andere getal gelijk is het
  + 25.00000 -voud van het eerste .|
asked: 2 numbers
applied substitutions:  1  1  1  1  3  11  35  41
x1 x2, waarvan verschil van derdemacht en tweedemacht van x2 is
+ 25.00000 -voud van eerste. is (verschil (x2 , x1 ),  + 1.00000 )|
applied references: 42
x1 x2 , waarvan verschil van derdemacht en tweedemacht van x2 is
+ 25.00000 -voud van x1 . is (verschil (x2 , x1 ),  + 1.00000 )|...
is (verschil (x2 , x1 ),  + 1.00000 ) is (verschil (derdemacht (x2 )
, tweedemacht (x2 )), produkt (  + 25.00000 , x1 ))|
x1 is x
```

```
solve(is (verschil (derdemacht ( som(      +1.00000  ,produkt(      +1.00000  ,
x))),tweedemacht ( som(      +1.00000  ,produkt(      +1.00000  ,x)))),
produkt (    +25.00000 ,x)),x, som(      +1.00000  ,produkt(      +1.00000  ,
x)), 0, false , false );
```

```
+ 4.0000     + 5.0000
- 6.0000     - 5.0000
  +.0000     + 1.0000
```

Notice how derdemacht *finds the right operand.*

Bepaal twee getallen ,waarvan de som 30 en het produkt 221 is. |

```
? twee getallen, waarvan empty som  + 30.00000
en het produkt  + 221.00000 is .|
twee getallen, waarvan som  + 30.00000 en het produkt  + 221.00000 is .|
asked: 2 numbers
applied substitutions:  1  27
x1 x2 , waarvan som  + 30.00000 en produkt  + 221.00000 is .|
applied references:
x1 x2 , waarvan som  + 30.00000 en produkt  + 221.00000 is .|
x1 x2 empty waarvan som  + 30.00000 en produkt (x1 , x2 )  + 221.00000 is .|
x1 x2 empty waarvan som (x1 , x2 )  + 30.00000
en produkt (x1 , x2 )  + 221.00000 is .|
waarvan som (x1 , x2 ) + 30.00000 en produkt (x1 , x2 )  + 221.00000 is .|
is (som (x1 , x2 ),  + 30.00000 ) is (produkt (x1 , x2 ), + 221.00000)|
x1 is x
```

solve(is (produkt (x, verschil(+30.00000 ,produkt(+1.00000 ,x))),
+221.00000),x, verschil(+30.00000 ,produkt(+1.00000 ,x)),
0, false , false);

```
+ 13.0000   + 17.0000
+ 17.0000   + 13.0000
```

Notice how the right operands are found.

Verdeel het getal 60 in twee delen,zodat de som van de tweedemachten der delen 2858 bedraagt. |

```
verdeel het getal  + 60.00000 in twee getallen, waarvan
empty som van empty tweedemachten van getallen  + 2858.00000 is .|
verdeel het getal  + 60.00000 in twee getallen, waarvan
som van tweedemachten van getallen  + 2858.00000 is .|
asked: 2 numbers
applied substitutions:  1  4  6
x1 x2, waarvan som van tweedemachten van getallen  + 2858.00000 is .
is (som (x1, x2),  + 60.00000 )|
applied references : 49
x1 x2, waarvan som van tweedemachten van x1 x2  + 2858.00000 is .
is (som (x1 ,x2 ),  + 60.00000 )|
x1 x2 , waarvan som van tweedemacht (x1) tweedemacht (x2)  + 2858.00000 is.
is (som (x1 , x2 ),  + 60.00000 )|
x1 x2, waarvan som (tweedemacht (x1 ), tweedemacht (x2 )) + 2858.00000 is.
is (som (x1 , x2 ),  + 60.00000 )|
waarvan som (tweedemacht (x1 ), tweedemacht (x2 ))  + 2858.00000 is .
is (som (x1 , x2 ),  + 60.00000 )|
is (som (x1 , x2 ),  + 60.00000 ) is (som (tweedemacht (x1 ),
tweedemacht (x2 )),  + 2858.00000 )|
x1 is x
```

```
solve(is (som (tweedemacht (x),tweedemacht ( verschil(      +60.00000  ,
produkt(       +1.00000  ,x))),  +2858.00000 ),x, verschil(
    +60.00000  ,produkt(       +1.00000  ,x)), 0, false , false );
```

```
+ 53.0000    +  7.0000
+  7.0000    + 53.0000
```

Notice treatment of plural tweedemachten.

De som van de vierkanten van drie getallen die met 2 opklimmen is 5555. Bereken die getallen. |

```
empty som van empty tweedemachten van drie getallen die met  + 2.00000
opklimmen is  +  5555.00000.? die getallen .|
som van tweedemachten van drie getallen die met  + 2.00000
opklimmen is  +  5555.00000. |
asked: 3 numbers
applied substitutions: 25
som van tweedemachten van x1 x2 x3 is  + 5555.00000 .
is (verschil (x3, x2 ), + 2.00000 ) is (verschil (x2 , x1 ), + 2.00000 )|
applied references:
som van tweedemachten van x1 x2 x3 is  + 5555.00000 .
is (verschil (x3 , x2 ),  + 2.00000 ) is (verschil (x2 , x1 ), + 2.00000 )|
som van tweedemacht (x1) tweedemacht (x2) tweedemacht (x3) is + 5555.00000 .
is (verschil (x3 , x2 ), + 2.00000 ) is (verschil (x2 , x1 ), + 2.00000 )|
som (tweedemacht (x1), tweedemacht (x2), tweedemacht (x3)) is + 5555.00000 .
is (verschil (x3, x2 ),  + 2.00000 ) is (verschil (x2 , x1 ), + 2.00000 )|
som (tweedemacht (x1), tweedemacht (x2), tweedemacht (x3)) is + 5555.00000 .
is (verschil (x3 , x2 ), + 2.00000 ) is (verschil (x2 , x1 ), + 2.00000 )|
is (verschil (x3 , x2 ), + 2.00000 ) is (verschil (x2 , x1 ), + 2.00000 )
is (som tweedemacht (x1), tweedemacht (x2), tweedemacht (x3)), + 5555.00000)|
x2 is x
```

```
solve(is (som3 (tweedemacht ( som(  verschil(0,      +2.00000 ) ,produkt(
     +1.00000  ,x))),tweedemacht (x),tweedemacht ( som(      +2.00000  ,
produkt(      +1.00000  ,x)))),  +5555.00000 ), som(  verschil(0,      +2.00000 )
,produkt(      +1.00000  ,x)),x, som(      +2.00000  ,produkt(      +1.00000  ,
x)), false , false );
```

```
+ 41.0000     + 43.0000     + 45.0000
- 45.0000     - 43.0000     - 41.0000
```

Notice finding of the right main unknown.

Van twee getallen is het een tweemaal zo groot als het andere;hun produkt is 288. Bereken die getallen. |

```
van twee getallen is het een tweemaal empty groot als het andere ;
hun produkt is  + 288.00000 .? die getallen .|
van twee getallen is het een tweemaal groot als het andere;
hun produkt is  + 288.00000 .|
asked: 2 numbers
applied substitutions:  1  5  17  18  27
van x1 x2 is eerste produkt van  + 2.00000 andere ;
hun produkt is  . 288.00000 .|
applied references:  42  43  45
van x1 x2 is x1 produkt van  + 2.00000 x2 ;
produkt van x1 x2 is  + 288.00000 .|
van x1 x2 is x1 produkt van  + 2.00000 x2 ;
produkt (x1 , x2 ) is  + 288.00000 .|
van x1 x2 is x1 produkt ( + 2.00000 , x2 ) ;
produkt (x1 , x2 ) is  + 288.00000 .|
is x1 produkt (  + 2.00000 , x2 ) ; produkt (x1 , x2 ) is  + 288.00000 . |
is (x1 , produkt (  + 2.00000 , x2 ))
is (produkt (x1 , x2 ),  + 288.00000 )|
x1 is x
```

```
solve(is (produkt (x, som(      +.00000  ,produkt(      +.50000  ,x))),
  +288.00000 ),x, som(      +.00000  ,produkt(      +.50000  ,x)), 0,
false , false );
```

```
+ 24.0000    + 12.0000
- 24.0000    - 12.0000
```

Notice treatment of tweemaal.

Twee getallen hebben 10 tot som. De som van hun derdemachten is 280.
Welke getallen zijn dat? |

```
twee getallen hebben  + 10.00000 tot som , empty som van hun derdemachten
is  + 280.00000 .? getallen zijn dat ?|
twee getallen hebben  + 10.00000 tot som . som van hun derdemachten
is  + 280.00000 .|
asked: 2 numbers
applied substitutions:  20  27
x1 x2 waarvan som is  + 10.00000 . som van hun derdemachten
is  + 280.00000 .|
applied references: 45
x1 x2 waarvan som is  + 10.00000 . som van derdemachten
van x1 x2 is  + 280.00000 .|
x1 x2 waarvan som is  + 10.00000 . som van derdemacht (x1) derdemacht (x2)
is  + 280.00000 .|
x1 x2 waarvan som is  + 10.00000 . som     (derdemacht(x1),derdemacht (x2))
is  + 280.00000 .|
x1 x2 waarvan som (x1 , x2 ) is  + 10.00000 .
som (derdemacht (x1 ), derdemacht (x2 )) is  + 280.00000 .|
waarvan som (x1 , x2 ) is  + 10.0000 , som (derdemacht (x1),derdemacht (x2)),
is  + 280.00000 .|
is (som (x1 , x2 ),  + 10.00000 ) is (som (derdemacht (x1),derdemacht (x2)),
    + 280.00000 )|
x1 is x
```

```
solve(is (som (derdemacht (x),derdemacht ( verschil(     +10.00000  ,produkt(
    +1.00000  ,x)))),   +280.00000 ),x, verschil(     +10.00000  ,produkt(
    +1.00000  ,x)), 0, false , false );
```

```
+ 6.0000    + 4.0000
+ 4.0000    + 6.0000
```

Notice treatment of hun.

De som der kwadraten van twee opvolgende gehele getallen is 1301.
Bepaal die getallen. |

```
empty som van tweedemachten van twee opeenvolgende geheel getallen is
+ 1301.00000 . ? die getallen .|
som van tweedemachten van twee opeenvolgende getallen   is + 1301.00000.|
asked: 2 numbers, Integer
applied substitutions: 3  11
som van tweedemachten van x1 x2 is  + 1301.00000
is (verschil (x2 , x1 ),  + 1.00000 )|
applied references:
som van tweedemachten van x1 x2 is  + 1301.00000
is (verschil (x2 , x1 ),  + 1.00000 )|
som van tweedemacht (x1 ) tweedemacht (x2 ) is  + 1301.00000
is (verschil (x2 , x1 ),  + 1.00000 )|
som (tweedemacht (x1 ), tweedemacht (x2 )) is  + 1301.00000
is (verschil (x2 , x1 ),  + 1.00000 )|
som (tweedemacht (x1 ), tweedemacht (x2 )) is  + 1301.00000
is (verschil (x2 , x1 ),  + 1.00000 )|
is (verschil (x2 , x1 ),  + 1.00000 )   is (som (tweedemacht (x1 ),
tweedemacht (x2 )),  + 1301.00000 )|
x1 is x
```

```
solve(is (som (tweedemacht (x),tweedemacht ( som(      +1.00000  ,produkt(
     +1.00000  ,x)))),  +1301.00000 ),x, som(      +1.00000  ,produkt(
     +1.00000  ,x)), 0, true , false );
```

```
+ 25.0000    + 26.0000
- 26.0000    - 25.0000
```

Notice condition Integer.

Zoek een getal,waarvan de tweedemacht 1 1/9 meer is dan het drievoud. |

```
? een getal, waarvan empty tweedemacht  + 1.11111 groter is dan het drievoud.|
een getal, waarvan tweedemacht  + 1.11111 groter is dan het drievoud .|
asked: 1 numbers
applied substitutions:  1  7  8  26
x1, waarvan tweedemacht is som van  + 1.11111  + 3.00000 - voud .|
applied references:
x1, waarvan tweedemacht is som van  + 1.11111  + 3.00000 - voud .|
x1 empty waarvan tweedemacht is som van  + 1.11111 + 3.00000 - voud(x1 ).|
x1 waarvan tweedemacht is som van + 1.11111 produkt ( + 3.00000 , x1 ).|
x1 waarvan tweedemacht is som ( + 1.11111 , produkt ( + 3.00000 , x1 )).|
x1 waarvan tweedemacht (x1) is som ( + 1.11111 , produkt ( + 3.00000,x1 )).|
waarvan tweedemacht (x1) is som ( + 1.11111 , produkt ( + 3.00000,x1 )).|
is (tweedemacht (x1 ), som ( + 1.11111 , produkt ( + 3.00000, x1 )))|
```

solve(is (tweedemacht (x),som (+1.11111 ,produkt (+3.00000 ,x))), x, 0, 0, false , false);

```
+ 3.3333
  -.3333
```

Notice treatment of voud.

Zoek een getal,waarvan de tweedemacht evenveel meer is dan 120,als het zelf minder is dan 120. |

? een getal, waarvan empty tweedemacht evenveel groter is dan + 120.00000 ,
als het eerste minder is dan + 120.00000 . |
een getal, waarvan tweedemacht evenveel groter is dan + 120.00000,
als het eerste minder is dan + 120.00000 . |
asked: 1 numbers
applied substitutions: 1 8 9. 26
x1, waarvan tweedemacht is som van evenveel + 120.00000,
als is verschil van + 120.00000 en eerste. |
applied references: 42
x1, waarvan tweedemacht is som van evenveel + 120.00000,
als is verschil van + 120.00000 en x1 . |
x1, waarvan tweedemacht is som van evenveel + 120.00000,
als is verschil (+ 120.00000 , x1) empty . |
x1, waarvan tweedemacht is som van verschil (+ 120.00000 , x1) + 120.00000,
empty .|
x1, waarvan tweedemacht is som(verschil (+ 120.00000,x1),+ 120.00000)empty.|
x1 empty waarvan tweedemacht (x1) is som (verschil (+ 120.00000 , x1),
+ 120.00000), empty . |
waarvan tweedemacht (x1) is som (verschil (+ 120.00000 , x1),
+ 120.00000), . |
is (tweedemacht (x1), som (verschil (+ 120.00000 , x1), + 120.00000))|

solve(is (tweedemacht (x),som (verschil (+120.00000 ,x), +120.00000)),
x, 0, 0, false , false);

+ 15.0000
- 16.0000
Notice treatment of evenveel.

Drie getallen verhouden zich als 1,2 en 4. De som van de vierkanten dier getallen is 525. Bereken die getallen. |

```
drie getallen verhouden zich als  + 1.00000 ,  + 2.00000 en  + 4.00000 .
empty som van empty tweedemachten dier getallen is
   + 525.00000 .? die getallen .|
drie getallen verhouden zich als  + 1.00000 ,  + 2.00000 en  + 4.00000 .
som van tweedemachten dier getallen is  + 525.00000 .|
asked: 3 numbers
applied substitutions: 2  24
x1 x2 x3 . som van tweedemachten van die getallen is  + 525.00000 .
is (produkt (x3 ,  + 2.00000 ), produkt (x2 ,  + 4.00000 ))
is (produkt (x2 ,  + 1.00000 ), produkt (x1 ,  + 2.00000 ))|
applied references: 48
x1 x2 x3 . som van tweedemachten van x1 x2 x3 is  + 525.00000 .
is (produkt (x3 ,  + 2.00000 ), produkt (x2 ,     + 4.00000 ))
is (produkt (x2 ,  + 1.00000 ), produkt (x1 ,     + 2.00000 ))|...
is (produkt (x3 ,  + 2.00000 ), produkt (x2 ,     + 4.00000 ))
is (produkt (x2 ,  + 1.00000 ), produkt (x1 ,     + 2.00000 ))
is (som (tweedemacht (x1), tweedemacht (x2), tweedemacht (x3)),+ 525.00000)|
x2 is x
```

```
solve(is (som3 (tweedemacht ( som(      +.00000  ,produkt(      +.50000  ,
x))),tweedemacht (x),tweedemacht ( som(      +.00000  ,produkt(
     +2.00000  ,x)))),   +525.00000 ), som(      +.00000  ,produkt(
+.50000  ,x)),x, som(      +.00000  ,produkt(      +2.00000  ,x)),
false , false );
```

```
+ 5.0000    + 10.0000    + 20.0000
- 5.0000    - 10.0000    - 20.0000
```

Notice treatment of verhouden zich als.

Welk getal is het 9-voud van zijn omgekeerde? |

```
? getal is het  + 9.00000 -voud van zijn omgekeerde ?|
getal is het  + 9.00000 -voud van zijn omgekeerde |
asked: 1 numbers
applied substitutions: 1
getal is  + 9.00000 -voud van zijn omgekeerde |
applied references: 44  46
x1 is  + 9.00000 -voud van omgekeerde van x1 |
x1 is  + 9.00000 -voud van omgekeerde (x1 ) |
x1 is  + 9.00000 -voud (omgekeerde (x1 ))|
x1 is produkt (  + 9.00000 , omgekeerde (x1 ))|
x1 is produkt (  + 9.00000 , omgekeerde (x1 ))|
is (x1 , produkt (  + 9.00000 , omgekeerde (x1 )))|
```

solve(is (x,produkt (+9.00000 ,omgekeerde (x))),x, 0, 0, false , false);

```
+ 3.0000
- 3.0000
```

The first of four school problems. Notice operator voud.

Twee getallen zijn samen 13. Hun middelevenredige is 6. Welke zijn die getallen? |

```
twee getallen zijn samen  + 13.00000 . hun middelevenredige is  + 6.00000
.? zijn die getallen ?|
twee getallen zijn samen  + 13.00000 . hun middelevenredige is  + 6.00000.|
asked: 2 numbers
applied substitutions:  16  27
x1 x2 waarvan som is  + 13.00000 . hun middelevenredige is  + 6.00000 .|
applied references: 45
x1 x2 waarvan som is  + 13.00000 . middelevenredige van x1 x2 is  + 6.00000.|
x1 x2 waarvan som is  + 13.00000 . middelevenredige (x1,x2) is  + 6.00000.|
x1 x2 waarvan som (x1 , x2 ) is  + 13.00000 .
middelevenredige (x1 , x2 ) is  + 6.00000 .|
waarvan som (x1 , x2 ) is  + 13.00000 .
middelevenredige (x1 , x2 ) is  + 6.00000 .|
is (som (x1 , x2 ),  + 13.00000 ) is (middelevenredige (x1,x2), + 6.00000)|
x1 is x
```

solve(ismiddelevenredige (x, verschil(+13.00000 ,produkt(+1.00000 , x)), +6.00000),x, verschil(+13.00000 ,produkt(+1.00000 , x)), 0, false , false);

```
+ 4.0000    + 9.0000
+ 9.0000    + 4.0000
```

The second of four school problems. Notice treatment of middelevenredige.

Verdeel 36 zo in twee delen , dat de som van de kwadraten der delen 336 meer is dan hun produkt. |

```
verdeel  + 36.00000 empty in twee getallen, dat empty som van empty
tweedemachten van getallen  + 336.00000 groter is dan hun produkt .|
verdeel  + 36.00000 in twee getallen , dat som van
tweedemachten van getallen  + 336.00000 groter is dan hun produkt .|
asked: 2 numbers
applied substitutions:  6  8  36
x1 x2 waarvan som van tweedemachten van getallen is som van
+ 336.00000 hun produkt . is (som (x1 , x2 ),  + 36.00000 )|
applied references:  49  45
x1 x2 waarvan som van tweedemachten van x1 x2 is som van
+ 336.00000 produkt van x1 x2 , is (som (x1 , x2 ),  + 36.00000 )|
x1 x2 waarvan som van tweedemachten van x1 x2 is som van
+ 336.00000 produkt (x1 , x2 ). is (som (x1 , x2 ),  + 36.00000 )|
x1 x2 waarvan som van tweedemachten van x1 x2 is som (  + 336.00000
, produkt (x1 , x2 )). is (som (x1 , x2 ),  + 36.00000 )|
x1 x2 waarvan som van tweedemacht (x1 ) tweedemacht (x2 ) is som (+336.00000
, produkt (x1 , x2 )). is (som (x1 , x2 ),  + 36.00000 )|
x1 x2 waarvan som (tweedemacht (x1 ), tweedemacht (x2 )) is som (+ 336.00000
, produkt (x1 , x2 )). is (som (x1 , x2 ),  + 36.00000 )|
waarvan som (tweedemacht (x1 ), tweedemacht (x2 )) is som (  + 336.00000
, produkt (x1 , x2 )). is (som (x1 , x2 ),  + 36.00000 )|
is (som (x1 , x2 ),  + 36.00000 ) is (som (tweedemacht (x1),tweedemacht (x2))
, som (  + 336.00000 , produkt (x1 , x2 )))|
x1 is x
```

```
solve(is (som (tweedemacht (x),tweedemacht ( verschil(      +36.00000  ,produkt
(       +1.00000  ,x)))),som (   +336.00000 ,produkt (x, verschil(
    +36.00000  ,produkt(       +1.00000  ,x))))),x, verschil(      +36.00000
,produkt(       +1.00000  ,x)), 0, false , false );

+ 20.0000     + 16.0000
+ 16.0000     + 20.0000
```

The third of four school problems.

Verdeel 36 in twee delen zo , dat het grootste van die twee delen middelevenredige is van het kleinste der delen en het oorspronkelijke getal. Welke delen zijn dat? |

```
verdeel  + 36.00000 in twee getallen empty, dat het andere van die twee
getallen middelevenredige is van het eerste van getallen en het eerste getal
.? getallen zijn dat ?|
verdeel  + 36.00000 in twee getallen, dat het andere van die twee getallen
middelevenredige is van het eerste van getallen en het eerste getal .|
asked: 2 numbers
applied substitutions:  1  1  1  6  27  34  36
x1 x2 waarvan andere van die x1 x2 middelevenredige is van eerste van
getallen en x1 . is (som (x1 , x2 ),  + 36.00000 )|
applied references:  43  52  42  49
x1 x2 waarvan x2 van x1 x2 middelevenredige is van x1 van x1 x2 en x1 .
is (som  (x1 , x2 ),  + 36.00000 )|
x1 x2 waarvan x2 van x1 x2 middelevenredige (x1 , x2 ) is van x1 van
x1 x2 en x1 . is (som (x1 , x2 ),  + 36.00000 )|
waarvan x2 middelevenredige (x1 , x2 ) is . is (som (x1 , x2 ),+ 36.00000)|
is (som (x1 , x2 ),  + 36.00000 ) is (x2 , middelevenredige (x1 , x2 ))|
x1 is x
```

```
solve(ismiddelevenredige (x, verschil(      +36.00000  ,produkt(       +1.00000  ,
x)), verschil(      +36.00000  ,produkt(       +1.00000  ,x))),x, verschil(
     +36.00000  ,produkt(       +1.00000  ,x)), 0, false , false );
```

```
+ 18.0000     + 18.0000
+ 36.0000         -.0000
```

The last of four school problems.
The solution is ***wrong;*** *notice that the reference for* oorspronkelijke *is not found correctly and that* middelevenredige *does not find the right operands.*

4.7. Evaluation of the program

To test the relevance of the program described, we would like to have word problems from an outside source, i.e. new to the program writer. The limitations on these problems are twofold:

1. They have to make use only of the 100-odd words that can be understood by the program.
2. They have to lead to quadratic equations in a way understandable by high school pupils.

Existing problems in text-books usually violate the first restriction. Therefore the members of the Computation Department of the Mathematical Centre were given a list of available words and asked to construct word problems from them. Although their problems were very useful in testing the procedures, they usually did not lead to equations that could be solved by the secondary program. It appears that mathematicians forget what problems they encountered in high school.

Luckily, one of our colleagues, Mr. Baanstra, was also teaching mathematics in high school. He made a set of four problems which were solved by his school class and by our computer. The text of these problems and their solutions can be found on pages 157-160.

For three correct solutions out of four problems our program would have earned the mark 7.5 (in a scale from 1 to 10), and done better than the highest class in the high school, which scored 5.2. The fourth problem was solved incorrectly by the computer. The referend of oorspronkelijke was not found to be the number 36 but one of the parts of it. It could be argued that this is also a possible interpretation of the stated word problem.

sources Chapter 4

[1] A.M. Turing, Can a Machine Think?, in: J.R. Newman (Ed.), The World of Mathematics pp. 2099-2123.

[2] Y. Bar-Hillel, The present state of automatic translation, in F.L. Alt (Ed.): Advances in computers, vol. 1, Academic Press, New York 1960.

[3] Language and machines, National Academy of Sciences, Washington, 1966.

[4] R.F. Simmons, Answering English Questions by computers - A survey. S.D.C. Report SP-1556, Santa Monica, California, April 1964.

[5] Automated Storage and Retrieval Program of Biomedical Information, Excerpta Medica Amsterdam.

[6] E.W. Dijkstra, Letter to the Editor, CACM 7, 3,March 1964, p.190.

[7] D.G. Bobrow, Natural Language Input for a Computer Problem Solving System, Ph.D. thesis, Mathematics Department, M.I.T., Cambridge (Mass.), 1964.

[8] D.G. Bobrow, A Question-answering System for High School Algebra Word Problems, Proceedings Fall Joint Computer Conference 1964, pp. 592-614.

[9] E.J. Wasscher, Nieuw Leerboek der Algebra, Tweede Deel, Wolters, Groningen 1958.

[10] Stoelinga and van Tol, Leerboek der Algebra Deel II, Tjeenk Willink, Zwolle 1958 (fifteenth edition).

[11] Birkenhäger and Machielsen, Nieuw Algebraboek 3, Noordhoff Groningen-Djakarta 1956 (eighth edition).

[12] F.E.J. Kruseman Aretz, Het MC-ALGOL 60-systeem voor de X8, MR 81, Mathematical Centre, Amsterdam 1966.

SAMENVATTING

Nu de taalkunde steeds exacter wordt, en de computer steeds toegankelijker voor de niet-specialist, kan de *Computational Linguistics* tot bloei komen. In dit proefschrift beschrijven we vier taalkundige onderzoekingen met behulp van een rekenautomaat.

In het eerste hoofdstuk wordt de opbouw van Nederlandse woorden onderzocht, waarbij de *lettergreep* centraal staat. ALGOL-programma's worden gegeven die lettergrepen tellen en scheiden (het laatste is van belang voor automatisch zetten). Dit maakt een kwantitatieve analyse van de spellingslettergreep mogelijk, op grond waarvan een generatieve grammatica van de Nederlandse lettergreep wordt gegeven (die b.v. merknamen kan produceren).

In het tweede hoofdstuk wordt de opbouw van samengestelde woorden onderzocht aan de hand van *getalnamen*. Een programma analyseert en synthetiseert hoofdtelwoorden in vier Westelijke talen en het Chinees, zodat vertaling mogelijk wordt.

In het derde hoofdstuk wordt de opbouw van Nederlandse zinsdelen onderzocht met als voorbeeld de *zelfstandignaamwoordsgroep*, waarvoor een context-vrije grammatika wordt opgesteld, die een automatische isolering van de zelfstandignaamwoordsgroepen uit Nederlandse tekst mogelijk maakt.

In het vierde hoofdstuk wordt de mogelijkheid onderzocht een computer een stuk tekst te laten begrijpen. Het begrijpen is operationeel te definiëren wanneer de tekst een *ingeklede vergelijking* is: de juiste oplossing moet gevonden worden. Een ALGOL-programma wordt gegeven dat ingeklede vierkantsvergelijkingen oplost.

Nog vele problemen met woorden, zinsdelen en zinnen wachten op een behandeling waarbij de computer een rol kan spelen, zowel om volumineus materiaal te beheersen, als om gecompliceerde situaties exact te beschrijven. Het is mede de bedoeling van dit proefschrift om met voorbeelden anderen hiertoe aan te zetten.