Centrum voor Wiskunde en Informatica

REPORT_RAPPORT_

SEN

Software Engineering

Software ENgineering

SEN

The Dijkstra-Zonneveld ALGOL 60 compiler for the Electrologica X1
historical note SEN, 2

F.E.J. Kruseman Aretz

NOTE SEN-N0301 JUNE 30, 2003

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-3711

The Dijkstra–Zonneveld ALGOL 60 compiler for the Electrologica X1

F.E.J. Kruseman Aretz

Abstract

In the summer of 1960 Edsger W. Dijkstra and Jaap A. Zonneveld put into operation the very first ALGOL 60 compiler in the world. Its code was never documented. This report contains the full assembly text of one of the latest versions of that compiler (from 1964).

In order to make that text more accessible, an equivalent version in Pascal is added, together with eight chapters introducing the compiler and explaining its major features.

2000 Mathematical Subject Classification

01-08, 68-03, 68N20

1998 Computer Science Classification

K.2, D.3.4

Keywords and Phrases

historical, ALGOL 60 compiler, Electrologica X1

Preface

The main purpose of this document is to preserve the code of what presumably has been the first working ALGOL 60 compiler. It was written for the Electrologica X1 by E.W. Dijkstra and J.A. Zonneveld at the Mathematical Centre in Amsterdam in the years 1959 and 1960. Its code has never been documented before.

Somewhere in the period 1962 to 1969, when I was working at the Mathematical Centre and was in charge of the maintenance of that ALGOL system, I started to type the full text of the compiler on a Friden Flexowriter, aiming to document the latest version of the compiler in a Mathematical–Centre report. Due to more urgent work and my departure from the institute it remained unfinished. Only after my retirement I was able to take up the project again.

Apart from presenting the compiler code in full, including its commentary in Dutch, much attention is paid to make that code accessible. This is done in two different ways. First, an equivalent Pascal version of the compiler code was written and is presented as well. Second, in a number of chapters the main components of the compiler are described and many aspects of the compiler are dealt with.

I am grateful for the hospitality of Philips Research Laboratories, where most of the work of preparing this document was carried out. I also gratefully used computer facilities at the Eindhoven Technical University. Critical comments of R.R. Hoogerwoord were very helpful to improve the readability of the text.

It would have been a pleasure to me to dedicate this work to my friends Edsger Dijkstra and Jaap Zonneveld, from who I learned so much of computing science. Alas, Edsger died shortly. So I can only dedicate it to Jaap and to the memory of Edsger.

Eindhoven, september 2002

Contents

Chapter 1

Introduction

This report documents the first ALGOL 60 compiler, written by E.W. Dijkstra and J.A. Zonneveld at the Mathematical Centre in Amsterdam in the period from november 1959 to august 1960. It was written for the Electrologica X1, a machine developed at the Mathematical Centre but built by a Dutch computer factory specially founded for that purpose.

Although Dijkstra wrote a few papers on the compiler [4, 6, 7] and although part of the total system was documented in reports of the Mathematical Centre, the compiler code itself never was fully described and documented. This report tries to remedy that situation. Its value is not the possibility to use the documented code on an X1 emulator (which can and has been done); nor will it influence the state of the art in compiler writing. Its value, if any, is purely historical: it is a report on the result of an undertaking that was new for that time, in spite of the existence of Fortran and Cobol.

ALGOL 60 was a tremendous step forward, a milestone in the development of computing as a science, and writing a compiler for a language with such a new and rich structure required the invention of many new techniques. The compiler text shows which solutions were found for the problems encountered. It also reveals the struggle with many problems. One of the most impressive facts is that the compiler had to work in a store of 4K 27–bit words, in which both compiler code and working space had to be embedded.

The X1 ALGOL 60 system became operational in august 1960 and was used at the Mathematical Centre until the late sixties.

This report presents the compiler text in full. It does so in the (rather primitive) assembly language of the X1, which in its turn is documented in Dijkstra's PhD thesis

[1]. Since that compiler text is not very accessible even for readers knowing Dutch and X1 assembly language, a more or less equivalent version of it in (standard) Pascal has been added. These compiler codes are preceded by eight chapters explaining the most important aspects of the compiler.

In the remaining part of this introduction we deal with some general aspects in more detail.

1.1 Some history

The Mathematical Centre of Amsterdam played an important role in the development of the 'Algorithmic Language ALGOL', later (with the publication of the 1960 Report [9]) called ALGOL 60. It was A. van Wijngaarden who took part in the group responsible for the language definition. This group was the cradle of the IFIP working group WG 2.1.

In the annual reports of the Mathematical Centre ALGOL is mentioned for the first time in the report on 1959. We cite[1]:

> Prof. Van Wijngaarden attended congresses and conferences on 'ALGOL' in Copenhagen, Paris and Mainz, [...]

In the annual report on 1959 we further find the following information:

> Prof. Van Wijngaarden and Dr. E.W. Dijkstra attended a congress on 'ALGOL' in Copenhagen. A congress on 'Information processing processes' in Paris was attended by Prof. Van Wijngaarden, J.A. Zonneveld, Dr. T.J. Dekker and M.L. Potters. In Mainz Van Wijngaarden gave a presentation on 'Divergent series', also attending there the so–called 'ALGOL' conference. F.J.M. Barning and Dr. T.J. Dekker took a course on 'ALGOL' in Darmstadt, [...]
>
> A research project that has the special interest of all staff members of the Computing Department is the one concerning the 'ALGOL'. In international context a draft is prepared of a universal language: 'ALGOL', i.e. ALGO–rithmic Language. This language shall be as close as possible to the standard notations in mathematics and be readable without further explanation. The language shall allow the description of any computational process, using the fixed algorithmic expressions, and it shall be translatable mechanically into machine programs. The definition of such a language is a big international project. The 'ALGOL' is now in 'statu nascendi';

[1]The annual reports were written in Dutch these years; translation by the author.

several national working groups are working towards its final shape and the international 'ALGOL' conferences organised regularly try to arrive at uniformity in notation of ALGOL programs; they do so under supervision of the international ALGOL committee. In this work the Computing Department makes an essential contribution. From about Oktober 1959 a team of five members of the department (A. van Wijngaarden, J.A. Zonneveld, E.W. Dijkstra, F.J.M. Barning and miss J.M. Feringa) are hard at work on the many problems presenting themselves here. As soon as the ALGOL language is cast in a definitive shape the construction of a compiler program for the electronic computer X1 can be turned to. This program shall be capable to derive, from a description in ALGOL, a program by which the calculations concerned can be executed on the X1.

Edsger Dijkstra, Bram Loopstra and Ria Debets in front of the Mathematical Centre building, 1954 (photograph G.A. Blaauw)

The 1960 annual report of the Mathematical Centre devotes a long passage to the ALGOL compiler:

The large–scale activity of the Computing Department with respect to ALGOL began already in November 1959. Due to the fact that Prof. Van Wijngaarden participated in the committee that, in January 1960, would decide on the final shape of ALGOL, there was ample reason to discuss the various aspects of algorithmic languages. The last two months of 1959 were also used to study the compiling technique as we learned it from Prof.Dr. H.D. Huskey and to subject it to a critical investigation.

Thus, when in January 1960 the final form of ALGOL – baptized 'ALGOL 60' in order to avoid confusion and as an expression of modesty of the composers – was stated, we already had a fair notion of the problems awaiting us. Moreover, we had the final data at our disposal at first hand, i.e. very rapidly. Largely due to these circumstances the ALGOL 60 compiler of the Mathematical Centre would be one of the first in the world, if not the very first one, that really did work. Possibly also the fact that precisely at that stage we got our own X1 at our disposal played a role: we were not yet accustomed to apply this machine in a certain manner and could therefore more easily start from scratch.

Because the implications of the language permeated to us only gradually we were not confronted with all problems at the same time and in a number of steps a closely fitting system was constructed. Then, in March, we had a three–day discussion in Copenhagen with a number of experts from Regnecentralen, intending to confront our ideas with theirs before starting the detailed elaboration. Our visit to Copenhagen resulted in a very important embellishment which we were able to incorporate in our projects within a couple of weeks. Immediately thereafter detailed elaborations started working in parallel projects. While Dr. E.W. Dijkstra and J.A. Zonneveld were developing the compiler Miss M.J.H. Römgens and Miss S.J. Christen started work on the organisational and arithmetic subroutines which should be at the disposal of the object program during its execution. Where the problems in the construction of the compiler were mainly of a logical nature, the work on the subroutines at the service of the object program were aggravated most by the requirement of maximal efficiency.

By July the compiler was subject to tests for the first time; a few weeks later object programs produced were actually executed for the first time. Most of the bugs that were revealed had the character of clerical errors or clear omissions (the latter especially in the compiler), for which the remedy was immediately obvious. Late September we had built up such strong confidence in our realization of ALGOL 60 that time was considered ripe for the organization of a course on 'Programming in ALGOL 60'. A syllabus was written and in November the first four–day course was given. Because of the overwhelming interest this course had to be repeated in December.

The great interest for these courses, the enthusiasm of the course–members and especially the good experiences with ALGOL 60 that the Computing Department

> has acquired itself for its own work confirmed us in the confidence that the labour invested in the completion of this project was not wasted. On the contrary!

So far our citation of the annual report 1960. Indeed it was an huge project for a computing department of 11 people. The compiler is about 2000 instructions long, another 2000 instructions support object–program execution. The latter 2000 instructions, constituting the collection of organisational and arithmetic subroutines supporting object–program execution, was baptized 'the Complex'. All these 4000 instructions were written (and tested) in no more than 9 months, quite a feat for a machine that was only put into use at the Mathematical Centre in March 1960.

The annual report of 1961 continues the interesting story of the ALGOL 60 project. We cite:

> Scientific activities of the Computing Department during 1961 largely concentrated on the ALGOL compiler that was finished in 1960. On account of the intensive use a number of further errors came to light (allbeit with decreasing frequency). Some of these were easily repaired, others, however, required quite an amount of brainwork.
>
> During the construction of standard procedures in ALGOL 60 it became apparent – after discussions in which eventually every member of the Computing Department would participate – that the formulation of standard processes is possible only in as far as the requirements to be met by the executing arithmetic are known. In concerted effort a number of such requirements were sketched. The arithmetic incorporated in 1960 appeared not to meet these requirements. A long list of small changes in the arithmetic complexes[2] proved necessary, changes that were carried through by Miss Römgens and Miss Christen with their usual precision.
>
> Once the implementation of these changes was decided upon, it was, for obvious reasons, given high priority. Hence the freshly started construction of an MCP–library (a library of standard procedures that the user can apply without prior declaration) was slowed down. What was, nevertheless, achieved in 1961 in MCPs, mainly by Mrs Goldschmeding–Feringa, concerned the control of the fast tape punch by ALGOL programs[3]. Besides the usual difficulties occurring while testing interrupt programs we were confronted here at the same time with the defects of the (yet untested) punch and its connection to the X1. It was therefore a great pleasure to see an ALGOL program producing tape one of the last days before Christmas.

[2]Originally there were two complexes of subroutines supporting object program execution: ALS, with single length floating point arithmetic, and ALD, with double–length arithmetic.

[3]When I entered the Mathematical Centre in 1962, only a slow tape punch (25 characters/sec) was connected to the X1. A fast one (300 characters/sec) was installed only in 1963.

Edsger Dijkstra and Jaap Zonneveld agreed not shave before the project of writing the ALGOL 60 compiler was done. Which, however, did not imply that they did shave when it was completed as scheduled August 24, 1960, 16:00 h. Zonneveld had a proper shave in March 1961 (picture from his personal archive); Dijkstra always kept his beard since.

> A few months were spent in writing two internal reports assessing the knowledge sofar available only by oral communication. These reports regard the construction of MCPs and the adaptation of the compiler and the complexes to other X1 installations[4]. They were written in order to be able to delegate these activities and to protect the Computing Department from the burden of these (mainly administrative) activities that are no longer of interest to it.
>
> [. . .]
>
> With several foreign visitors (both from universities and industry) the problem of implementing ALGOL 60 for their specific machines was discussed in various degree of detail.

The annual report of 1962 adds:

> For the ALGOL 60 compiler for the X1, finished in 1960, the construction of a library of standard procedures (series AP) was started. Several issues have been published in 1962.
> By these provisions the ALGOL system showed to be a highly serviceable system, not only for testing and theoretical purposes, but also for production.
> After installment of the system about 20% of machine time of the X1 was allocated for the execution of ALGOL programs. By the middle of 1962 this percentage had

[4]The first of these reports is presumably [5]; I never saw the other one.

> risen to a good 70%. The programming of procedures in machine code of the AP series (series AP 100) was performed by the staff members Mrs. Goldschmeding–Feringa, Miss Römgens, and Miss Christen under supervision of Mr. Dekker.
>
> Thanks to the fast growth of the ALGOL system, the department was able to spend more time on the investigation of numerical methods during 1962. The arithmetic complex with new, improved arithmetic, designed in 1961, was finished early in the year and put into operation February 1st.

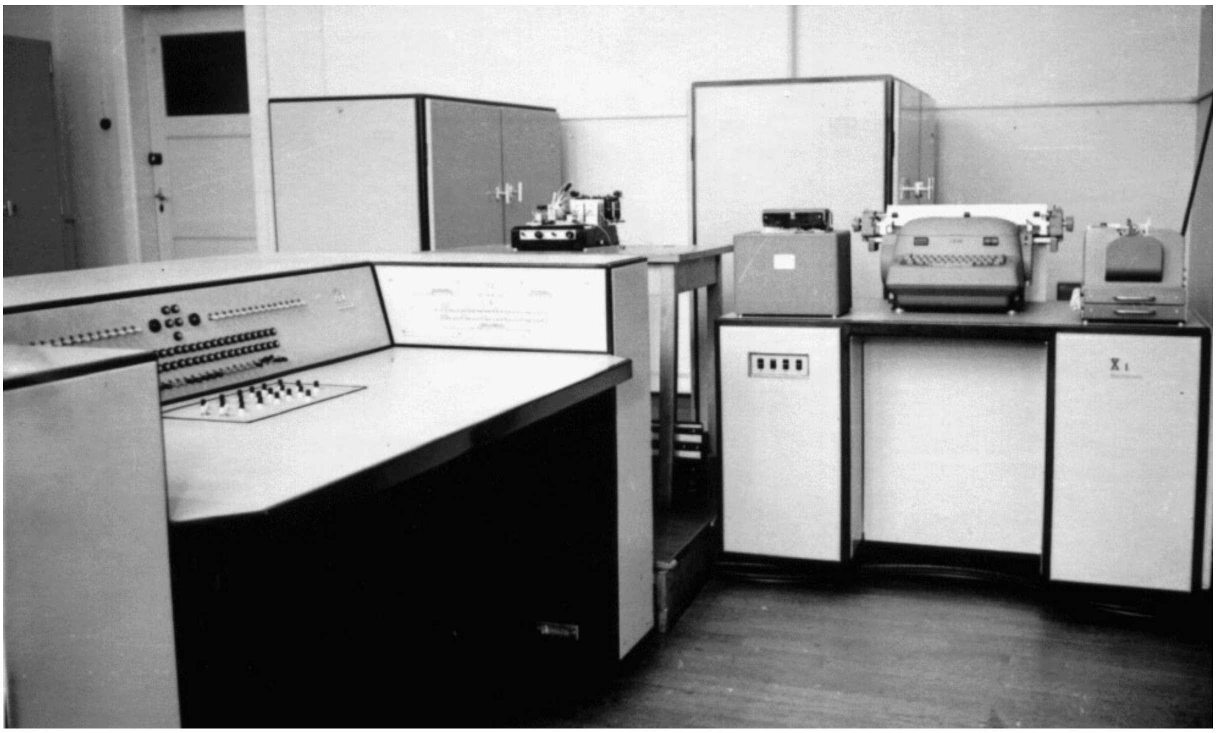
Some 31 machine code procedures (MCP's) were published that year in the series AP 100 and some 24 procedures in ALGOL 60 in the series AP 200. Moreover, the complexes ALD and ALS were printed as the series P (1)200. Also some manuals were released, in particular for working with ALGOL programs.

A year after completing the compiler, the ALGOL 60 system for the X1 was considered complete and no further developments to the compiler or to the complexes were planned. The key players embarked on new endeavours. Dijkstra left the Mathematical Centre in August 1962 for a chair at the Technical University Eindhoven. Zonneveld returned to his specialism, numerical analysis, and was now investigating Runge–Kutta methods for the numerical integration of differential equations, the subject of his PhD thesis[11] in 1964. When I joined the Mathematical Centre in September 1962 the original crew of the ALGOL 60 project for the X1 was almost dissolved.

1.2 The Electrologica X1

The Mathematical Centre had developed and built several automatic computers (ARRA and ARMAC) before it started the development of the X1. The latter project was soon to be continued by a commercial company founded for that purpose, Electrologica. This was a full subsidiary of a Dutch insurance company, Nillmij. The first design of the X1 had been completed by the end of 1956. It was a rather modern design. It was one of the first fully transistorized machines, it had an interrupt system, and an index register. Below we give some more details of the X1. A rather good description of its instruction repertoire and of its assembly language can be found in Dijkstra's PhD thesis[1]. An overview of the X1 is given in [8].

The X1 had a word and instruction length of 27 bits. It had two 27–bit registers called A and S, a 16–bit index register B, and some 1–bit registers, the most important of which was the condition register C. The instruction counter was called the T register.



The machine on which the work was done, the Electrologica X1 computer at the second floor of the Mathematical Centre.

It had integer arithmetic only. The number system was one–complement. It had some double–length instructions, in which registers A and S operated as one double–length register. This was the case in the (integer) multiplication and division instructions and in some of the shift instructions. Integer arithmetic was minus–zero preferent.

There was neither floating–point hardware nor support for a stack: all such operations had to be carried out completely by software. Also support for dynamic (i.e., two–level) addressing was absent.

The 27 bits of an instruction were, in general, structured in the following way:

3 bits	'function letter', indicating mostly the register involved
3 bits	'function digit', specifying the operation
2 bits	'A/B/C variant', giving the addressing mode
2 bits	'P/Z/E variant', specifying condition setting
2 bits	'U/Y/N variant', specifying condition following
15 bits	'address part', mostly specifying an address or a number

For register A (‘function letter’ 0) the following instructions[5] were available:

notation	meaning
0A n	A:= A + M[n]
1A n	A:= A − M[n]
2A n	A:= M[n]
3A n	A:= − M[n]
4A n	M[n]:= M[n] + A
5A n	M[n]:= M[n] − A
6A n	M[n]:= A
7A n	M[n]:= − A

The system here should be clear. Calling the function digit f we have:
– for $f < 4$, the destination of the result is the register (A), otherwise the word of memory (M[n]) involved;
– for $f < 2 \bmod 4$, the result is formed by addition of register and memory word, otherwise by taking register or memory word;
– for odd f, the inverse of the (second) operand is used rather than the operand itself.

Register S (‘function letter’ 1) and B (‘function letter’ 4) had analogous instructions.

For register T (’function letter’ 5), the instruction counter, we had:

notation	meaning	condition
0T n	T:= T + 1 + M[n]	
1T n	T:= T + 1 − M[n]	
2T n	T:= M[n]	
3T n	T:= − M[n]	
4T n m	M[m]:= M[m] − 1; T:= n	$(0 \leq m \leq 7)$
6T n m	M[m+8]:= T + 1; T:= n	$(0 \leq m \leq 15)$

Here 0T and 1T are jump instructions, 2T and 3T (indirect) goto instructions, 4T is a counting (direct) goto, whereas 6T is a subroutine call.

The function letters X, D, Y, Z, and P were used for multiplication (X), division (D), and a great number of special instructions. 0P, . . . , 3P denoted register–shift instructions.

There were logical instructions too, denoted with the function letter combinations LA and LS. ‘`0LA n`’ meant bit by bit ‘or’ between A and M[n], ‘`2LS n`’ bit by bit ‘and’ between S and M[n]; the function digits 1 and 3 implied as usual inversion of the second operand.

[5]Strictly speaking, the X1 assembler required, for technical reasons not to be discussed here, a notation ‘`0A n X 0`’ rather than ‘`0A n`’

The address part normally indicated a 15–bit store address.
In case of the A variant of the addressing mode it indicated a 15–bit natural number. Thus '`1B 1 A`' had as effect '`B:= B - 1`' and '`2T n A`' meant '`T:= n`', i.e. a (direct) transfer of control to (the instruction at memory cell) n.
In case of the B variant the contents of B were added to the address part before executing the instruction. Thus '`2A n B`' meant '`A:= M[B+n]`'. The addition '`B+n`' was carried out in 15 bits without end–around carry; '`B+32767`' had the effect of '`B-1`'.

Condition setting was done by means of the P/Z/E variants. The P variant set the condition register C affirmative if the result of the operation was positive, i.e. +0 or larger; the Z variant set C affirmative if the result of the operation was +0 or −0. Thus the instruction '`3A 0 A P`' had '`A:= -0; C:= No`' as effect.

Condition following was done by the Y/N variants. The Y variant caused the instruction to be executed only if the condition register was affirmative, otherwise it was skipped. The N variant required C to be negative for the instruction to be executed. The following instruction pair could be used to load the absolute value of '`M[n]`' into A:

```
  2A n P
N 3A n
```

The fact that condition following was available to all instructions and not to jump instructions only, lead often to compact code, the more so as the condition setting could have occurred many instructions before.

The U(ndisturbed) variant suppressed the assignment of the result of an operation to its final destination. It was used for condition setting without disturbing register or store. The instruction '`U 1A n P`' did not more than '`C:= (A > M[n])`'. The U variant could not be combined with each instruction.

The read–and–rewrite cycle of the core store was 32 μsec. Skipping an instruction took 32 μsec, instructions like '`2A n A`' (without a store operand) 36 μsec, instructions like '`0A n`' (with a store operand) 64 μsec, whereas multiplication and division took 500 μsec. On the average the X1 executed 20K instructions per second.

In the (rather primitive) assembly language addresses were specified relative to so–called 'paragraphs', indicated by two paragraph letters, formed with the 13 letters Z, E, F, H, K, L, R, S, T, W, U, Y, and N. The address '`n ZE m`' meant '`(32*m + n) ZE 0`', i.e. $32 * m + n$ places further than the address assigned to the paragraph–letter combination 'ZE'. The meaning of the paragraph–letter combinations were defined at the beginning of the X1 program. The letters X, D, and C were used without a second paragraph letter and had a fixed meaning: X = 0, C = 16384, and D = 245766. The text '`DP RZ 0X7`' defined RZ to mean address 224 (i.e. $7 * 32 + 0$).

The X1 had no operating system. It had two states, running or stopped. When running it could be stopped by turning a switch (Stop Next Instruction) or by setting a stop address in a number of toggles. It also stopped by executing a stop instruction. When stopped it could be started by pressing a button. Button 1 started the assembler which was present in read–only store (addresses from '`0 D 0`').

At the Mathematical Centre the X1 was installed in 1960 and put into daily use March 8th, 1960. Its (read/write) store was extended from 8K to 12K words in May, 1962. It had no backing store whatsoever (apart from paper tape). Originally it had a console typewriter, a tape reader and a tape punch as sole peripherals; later a fast tape punch and a plotter were connected.

1.3 Working with the ALGOL system for the X1

Nowadays, with backing stores of Gbytes even for the smallest PC, with on–screen editors and cheep laser printers it is hard to imagine how primitive (but exciting) life was these days.

It was a major improvement that ALGOL programs could be punched on a (Friden) Flexowriter, which produced, apart from the tape, also a print on paper[6]. It could (also new!) be used for text editing, by reading (and thus reprinting and repunching) the tape, inserting the changes at the right places.

The ALGOL system was contained in 5 system tapes: the compiler tape, the complex tape, the loader tape, the cross–reference tape and the library tape (the latter 4 tapes existed in two versions, for single–length and double–length arithmetic respectively). During the compilation process the compiler was, at least in principle, not overwritten. During object–program execution the complex was, at least in principle, not overwritten. Therefore it was possible to compile a number of ALGOL 60 programs in sequence after loading the compiler once, and to execute a number of object programs (using the same arithmetic) after loading one of the complexes. In practice this was done only rarely: programs were compiled and immediately executed most of the time.

In that case the compilation and execution of a (correct) ALGOL 60 program required the reading (and subsequent rewinding) of the following tapes:

1. the compiler tape,

[6]When I entered the institute there were already 4 (sic!) of these.

2. the tape(s) containing the ALGOL 60 program,
3. the tape(s) containing the ALGOL 60 program a second time (during the reading of this tape the object–program tape was punched),
4. the complex tape,
5. the loader tape,
6. the second part of the object–program tape (produced in step 3),
7. the cross–reference tape,
8. the first part of the object–program tape (produced in step 3),
9. the library tape,
10. the input–data tape(s).

If an ALGOL program did not use any of the library routines the reading of cross reference and library could be skipped; if a program had no input the last step had to be skipped. The reading of each tape had to be started by pushing one of the console buttons.

The greatest shortcoming of the system, however, was the almost complete absence of syntax checks and run–time checks. At compile time most checks had to do with the representation of the basic symbols on tape (mistrusting the proper functioning of the Flexowriter punch and the X1 tape reader) and with store management; there was also a check on undeclared identifiers. The run–time checks involved the arithmetic (especially integer overflow) and again the lexical level of the input tape, but did not cover stack overflow or array indices out–of–bound. A complete list of the error–stop numbers is given in Appendix A.

In case of a compile–time stop the operator could give as feed back to the programmer only the error number and the list of identifiers typed on the console typewriter[7] and could mark the position of the source tape in the tape reader at the moment of the stop. Even the error stop for an undeclared identifier did not mention that identifier!

Also in case of a run–time stop an error number was returned to the programmer, together with the output produced thusfar. There was no program debugger available. In case of erroneous results the only means of debugging was to recompile and rerun the program

[7]During the second reading of the source text the identifiers of labels, procedures and switches were typed when processing their declaration started.

with more output statements for intermediate results added to the source text. The stepwise execution of an ALGOL object program, using the start and stop buttons of the console, required, apart from a lot of machine time, an enormous knowledge of details of the ALGOL system and was used only in exceptional cases, for otherwise unsolvable problems and in cases where the correct functioning of the ALGOL system itself or of the X1 hardware was in doubt.

In 1963 a second ALGOL 60 system, developed by Nederkoorn and Van de Laarschot, became available. Although it was hardly used as complete system the compiler came in use as separate syntax checker (suppressing the punching of an object tape). In later years no (fresh) ALGOL program was run with the Dijkstra/Zonneveld system without a prior syntax check by the Nederkoorn/Van de Laarschot compiler.

The following ‘special properties of the MC–Algol–system’ (mostly restrictions) were mentioned in the user manual[8]:

1. Comments starting with ‘comment’ and ending by ‘;’ are permitted also at the beginning of the program. Apart from this a program shall have the form of an unlabelled block or an unlabelled compound statement, in other words start with ‘begin’ and end with ‘end’.
   After the last symbol ‘end’ the compiler does not accept any symbol to be skipped but requires a symbol ‘Carriage Return’.

2. In the series of symbols that are skipped after an ‘end’ symbol (not being the last one of the program) the symbols ‘begin’, ‘comment’, and the stringquotes ‘≮’ and ‘≯’ are not permitted.

3. Only the first nine symbols of identifiers do matter.

4. The following rules apply to numbers occurring in an Algol program:
   The number zero is interpreted always as being of type integer, even if a decimal point is included or a numeric part = 0 is followed by an exponent.
   A number that, because the absence of a decimal point and an exponent, is of type integer according to the rules is treated as being of type real as soon as its absolute value exceeds 67108863.
   The decimal exponent shall not exceed 600 in absolute value.

5. In Algol 60, function procedures can be called not only in expressions but also as a statement by themselves. In that case the function value is of no interest and will

[8]taken from the user manual dated December 12th, 1962; translation by the author.

be ignored. For the standard functions mentioned in Sections 3.2.4 and 3.2.5 of the Report and for 'read' and 'XEEN', however, holds that they may not be called as statement by their own in the MC–Algol–system.

6. The value of the standard function 'abs' has the same type as its argument. The standard function 'entier' may have an argument of type integer. The standard functions 'sqrt' and 'ln' operate on the absolute value of the argument.

7. The primaries of an expression are evaluated in left–to–right order. (We mention this in so many words because the Algol–60 report is suggesting it but does not settle it explicitely.)

8. Labels beginning with a digit are not permitted.

9. It is not permitted to embrace a block lexicografically by more than 30 blocks. Herein do count for–statements, procedure bodies, and actual parameters consisting of more than a single identifier or number also as blocks.

10. In a goto–statement the evaluation of any possible switch designator shall result in a well–defined value (label). If not so then the goto–statement is not equivalent to a dummy statement but undefined.

11. Not only the value of the controled variable – called 'V' below –, but also the identity of V (i.e. if it is an subscripted variable) may be changed in the statement following the for–clause. In the expressions occurring in a for–clause (i.e. between for and do), not only in the expressions in the list elements but also in any possible index expression of V, the call of function procedures with side effects should be avoided. Also it should be avoided that the identity of V depends on the value of V (e.g. a controled variable of the form A[A[1]]).
In a for list element of the form 'A step B until C', where A, B, and C denote arithmetic expressions, one should avoid the value of sign(B) to depend on the value of V. For in the MC–Algol–system the expression B is evaluated only once per cycle and already calculated for the first time before the assignment V:= A.

12. Upon a for–clause no conditional statement shall follow. In other words, '*do if*' is prohibited.

13. Only a comma symbol is permitted as parameter delimiter.

14. Except for the explicit prohibition for certain procedures it is allowed to present an actual parameter of type integer for a formal parameter specified as real (or vice versa) in a procedure statement or a function designator.
15. Declarations starting a block and specifications in a procedure declaration shall be given in the following order:

    1) scalars (<type> or own<type>) and strings

    2) arrays

    3) destinations (label or switch)

    4) procedures
16. Procedures in which declarations marked by the symbol 'own' occur function not in the official manner when used recursively.
17. Only integer numbers, possibly preceded by a sign symbol, are permitted as array bounds in array declarations of the outermost block or those preceded by 'own'.
18. The MC–Algol–system does not discriminate between 'real' and 'integer' as the first symbol of a function declaration: in each invocation the type of the result is determined by the arithmetic that is carried out this time.
19. The MC–Algol–system requires a specification for each formal parameter of a procedure declaration.
20. Procedure bodies starting with a label should be avoided.
21. A formal parameter specified as label or <type> procedure shall not occur in the value list.
22. Parameters in the value list are evaluated at procedure entry in the order of specification. (This is of importance when the evaluation of an actual parameter can influence the value of another one.)
23. An array in the value list may have at most five indices.

The restrictions contained in these 'properties' seldom gave any problem for the use of ALGOL 60 as a programming language. The generality of the implementation, including full block structure, recursive procedures, and name parameters, even Jensen's device, often lead to compact and nice algorithms.

To give an impression of the excution speed of ALGOL 60 programs on the X1 we collected the execution times of some statements in Figure 1.

statement	time
`i:= 1`	2.0 msec
`i:= i + 1`	3.0 msec
`A[i]:= 1`	5.0 msec
`y:= sin(x)`	26.5 msec
`p`	3.5 msec
`q(x)`	8.5 msec
`r(x)`	11.8 msec
<u>for</u> `i:= 1` <u>step</u> `1` <u>until</u> `1000` <u>do</u>	7650 msec

(in the context of the following declarations:
<u>integer</u> `i;` <u>real</u> `x;`
<u>procedure</u> `p; ;`
<u>procedure</u> `q(z);` <u>real</u> `z; ;`
<u>procedure</u> `r(z);` <u>value</u> `z;` <u>real</u> `z; ;`
)

Figure 1: execution times of some statements

The table clearly shows the trade–off between ease of programming in ALGOL 60 and execution speed. Incrementing an integer variable by one (cf. the second example in Figure 1) could be coded in X1 machine code in two instructions:

```
2A 1 A
4A n
```

executing in less than 100 μsec. The programmer himself has to locate the variable in memory and to choose what register to use for the operation. In ALGOL 60, on the other hand, he simply writes '`i:= i + 1`' without bothering about the way of execution. The variable `i` is located by the compiler and even the use of the variable in a recursive procedure is no problem at all. The price paid for this convenience is a slowing–down of the execution, in case of the X1 from some 100 μsec to about 3000 μsec, by the execution of 7 instructions of the object program and 56 instructions of 4 'operators' coded in 'the Complex' of administrative and arithmetic subroutines supporting object–program execution. In general the ease of programming in ALGOL 60 was paid by a loss of execution speed by a factor of 10. Given the fact that within two years more than 70% of machine time of the X1 at the Mathematical Centre was used for the compilation and execution of ALGOL 60 programs, the users were quite willing to pay the price.

1.4 Developments of the ALGOL system after 1962

The main developments of the ALGOL 60 system for the X1 after 1962 were the introduction of a load–and–go version of the system and the incorporation of a plotter. Moreover the MCP library was extended with some new routines and some checks were added, both at compile time and at run time.

The load–and–go version, in operation from autumn 1963, reduced greatly the tape handling. There was only one system tape, ALGOL source programs were read physically once only, and no object tape was punched at all. The development of this system was made possible by the much larger size of the store, 12K words instead of 4K for which the original version was written. (In 1965, also an 8K version of the load–and–go system was made on behalf of the University of Utrecht; then, the system had to be divided over two tapes, the second of which to be loaded after compilation of the ALGOL program.) Since during the loading phase of the compiler, part of the compiler was overwritten by the object program, however, the system tape had to be read for each ALGOL program anew. The new system facilitated a fast service with many small student programs for the Universities of Amsterdam.

A Calcomp plotter was connected to the X1 in 1964. A nice package of MCPs for driving it was developed by van Berckel and Mailloux and documented in [12].

For a very simple but effective partial check on the syntactical correctness of source programs counts of yet unpaired round and square brackets were added to the lexical scan routines. In the first compiler scan it was then checked whether these counts were both zero at the occurrence of a semicolon or end–symbol.

An equally simple, incomplete but rather effective check was added at run time. It was checked that the address of an array element lay within the area reserved for that array (for one–dimensional arrays this meant a complete index–out–of–bound check). This check could be easily added to the indexer routine of the complex without any further change of the system. Many of the first 'victims' got angry and requested to run their programs with the 'old' system!

Further improvements were made in the tape–reading routines such that tape reading was accelerated quite a lot.

But all these changes had in common that they affected the system only skin–deep: the heart of the system remained untouched.

In 1966 the X1 got the Electrologica X8 as competitor. Since the ALGOL 60 system on that machine ran about 100 times faster than the one on the X1, and since it had rather complete syntax and run–time error checks, the main stream of ALGOL 60 programs was directed to the X8 very soon. The X1 remained in use at the Mathematical Centre, however, until mid 1972.

1.5 The Pascal version of the compiler

The Pascal version of the compiler is written in ISO Standard Pascal. It is reverse–engineered rather close to the machine–code version. It has been tested thoroughly: for a range of ALGOL 60 programs it produces exactly the same object code as the original version in X1 code.

Being close to the original, there are, however, from sheer necessity, some differences. In machine code one can do things that are impossible in any higher level language.

First of all, the order of the subroutines is different, and much more systematic than in the machine–code version. We also used the structuring that Pascal permits: most of the procedures are local to one of the three main procedures: '*prescan*', '*main_scan*', and '*program_loader*'. In the machine–code version these parts are mixed up criss–cross. In order to facilitate the relation between the two texts we added to most parts of the Pascal text the paragraph letters of the corresponding machine–code part as a comment.

Second, in the machine–code version all variables were accomodated in store. Most simple variables had an address of the form '`n ZE m`' (with $m \in \{0, 1, 2\}$) or '`n RE 0`'. In the Pascal version these variables are just global or local variables in the program. On the other hand all lists maintained in store are allocated in the Pascal program in an array '*store*', modelling the store, with bounds 0 and 12287 as in the X1 of the Mathematical Centre.

Next, the X1 code contains a number of constant tables in the text, e.g. for the decoding of Flexowriter punch code, for the compact coding and decoding of object instructions, and for the prefill of the symbol table. These are partly accomodated in arrays (which then have to be given a value at run time by a piece of program code), and partly implemented by means of a case construct or by program text only: in initializing the symbol table just before invoking procedure *main_scan* instead of copying a table using a loop now the appropriate values are filled in by linear code.

In the X1 code the only means of transfering control is the jump instruction[9]. We tried to

[9]In many simple cases of conditional constructs also the condition following variants of the X1 were used.

make the text slightly more structured by using '**if** . . . **then** . . . **else** . . . ', '**while** . . . **do** . . . ', and '**repeat** . . . **until** . . . ' whereever simple.

Subroutines with multiple entry points also caused some problems. Some could be solved by splitting the subroutine into several separate subroutines. In one case (in the loader) where a subroutine conditionally added 1 or 2 to its link and where the subroutine call was followed by two jump instructions to cater for the normal exit and one of the exceptional cases we eliminated the whole subroutine.

But in general we believe that the Pascal version is a faithful and honest representation of the original machine code. It reveals that the style of programming has changed largely in the years since 1960, not the least by the activities of one of the primary authors of the X1 system.

1.6 The X1–code version of the compiler

When I entered the Mathematical Centre in 1962, there were two handwritten manuscripts (in pencil) of the compiler code, one of Dijkstra and the other of Zonneveld. They contained the original version of the compiler. This version differs from the text given in the present report – the load–and–go version of the compiler – in some well–isolated areas. Especially the parts 'fill result list' (FRL, paragraph letters ZF), 'read next symbol' (RNS, ZY), 'next ALGOL symbol' (NSS, HT), and 'read flexowriter symbol' (RFS, LK) differ, whereas the routines with paragraph letters LL upto SZ, which have to do with the load–and–go aspects, are totally absent in the original version. Dijkstra's copy was recently found again and is now available in the archives of CWI.

All changes and improvements made from 1962 were written in an exercise book much in the same way as the original version. After completing the load–and–go version of the compiler I felt the need to produce a complete text of the compiler in its new state; so I started to type it – for the very first time! – on Flexowriter. That code text was completed just before I left the Mathematical Centre in 1969, but I never had time to extend it to a full documentation.

After my retirement I decided to resume that documentation project. I retyped the Flexowriter print, now as a file in ASCII in my work station, profiting of all modern text–editing facilities. In order to have more than a visual check I wrote an X1 emulator, typed in the complex of run–time support routines, and was so able to rerun the X1 ALGOL 60 system. This made it also possible to check the outputs of the X1–code and the Pascal version of the compiler against one another and to carry out a number of measurements.

Those measurements would have been quite a job in the sixties, but with today's tools they were mere child's play.

Chapter 2

Overview

The ALGOL 60 compiler for de El X1 uses two text scans for producing the translation. Originally, the source text, punched on papertape in Friden Flexowriter code, was physically read twice. The two compiler scans, called prescan and main scan, used the same routines for scanning the text. Those routines constitute the lexical scan part of the compiler. A later version of this lexical scan stored its intermediate results during the prescan and retrieved these from store during the main scan.

The output of the main scan was originally punched on paper tape. The output tape consisted of three parts: the object code proper, still in a free locatable format, the constant list, containing all numbers that occurred in the ALGOL text, and the future list, containing the final destinations of all forward references. The object code was punched during the main scan itself, the two other parts at the end of the main scan. A special loading program was used to convert the object tape into executable code. In a later load–and–go version the output of the main scan was stored in memory without punching. The loading phase was executed immediately after completion of the main scan.

Chapter 3 discusses the lexical scan routines. Chapter 4 presents the prescan program. In Chapter 5 many aspects of the main–scan program are analysed. Chapter 6 gives an overview of three versions of the compiler output. Chapter 7 introduces the library system. Chapter 8 treats three versions of program loading. Finally, in Chapters 9 and 10 the Pascal version and the X1–code version of the compiler are printed in full.

The compiler does not use any of today's parsing methods. In fact, there is hardly any parsing at all, in the sense of checking whether the program text conforms the grammar rules and constructing the parse tree. Almost any text is 'accepted' and the inspection of the symbols constituting the text is merely done for the immediate production of the translation. There is, however, some resemblance with methods based on precedence grammars.

```
DDEL "["                                    EEO    29-4-60
                                                   12-8-60

        DA 0 E E0  DI
 0      2A 18 Z E0  Z          EFLA=0?
 1    Y 6T 0 K No 2    ⇒    RLA
 2      2A 1       A
 3      6A 0 Z E0          OFLA:=1
 4      2A 0       A
 5      6A 5 Z E0          OH1:=0
 6      2S 18 Z E0
 7      6T 0 Z T0 0    ⇒   FTL(EFLA)
 8      2S 1 Z E0
 9      6T 0 Z T0 0    ⇒   FTL(IFLA)
10      2S 4 Z E0
11      6T 0 Z T0 0    ⇒   FTL(MFLA)
12      2S 19 Z E0
13      6T 0 Z T0 0    ⇒   FTL(FFLA)
14      2S 2 Z E0
15      6T 0 Z T0 0    ⇒   FTL(JFLA)
16      2S 22 Z E0
17      6T 0 Z T0 0    ⇒   FTL(NID)
18      2A 1       A
19      6A 18 Z E0
20      6A 1 Z E0          EFLA:=IFLA:=1
21      2A 0       A
22      6A 4 Z E0          MFLA:=0
23      6T 0 Z W0 0    ⇒   FTD
24      2S 2 Z E0   Z      if JFLA=0
25    Y 6T 0 Z H0 1    ⇒   GAI
26      2T 0 E L0 A    ⇒
```

A page from Dijkstra's handwritten version of the compiler. See page 173.

Chapter 3

The lexical scan routines

After its revision in 1963 the lexical scan consists of four hierarchically linked routines, called *read_flexowriter_symbol* (RFS), *next_ALGOL_symbol* (NSS), *read_next_symbol* (RNS), and *read_until_next_delimiter* (RND).

The lowest level routine in the hierarchy is *read_flexowriter_symbol*. The Flexowriter code has two shifts, lower case and upper case, with explicit punchings marking shift changes. Therefore, RFS keeps the most recent shift in the variable *rfsb*. RFS reads one or more punchings from the input tape, skips blank and erase codes, records shift punchings, checks parity and delivers as function value the next relevant code in internal representation.

The next level routine in the hierarchy is *next_ALGOL_symbol*. Its main task is to assemble basic symbols that are represented by more than one Flexowriter symbol, such as word delimiters, colonequal, unequal, or string quotes. Moreover it skips – outside strings! – comments introduced by the basic symbol '**comment**' and closed by a semicolon. Symbols between a basic symbol '**end**' and the next semicolon, '**end**', or '**else**'[1] are, however, not skipped by NSS and only ignored by the prescan and – once again – by the main scan.

The third level routine in the hierarchy – nonexistent originally – is *read_next_symbol*. During prescan it calls NSS for the next symbol and assigns it to the variable *dl*. Moreover it stores that symbol in a symbol store, packing three symbols in one computer word. During the main scan it takes its symbols from the symbol store and assigns them to *dl*.

The upper level of the hierarchy is routine *read_until_next_delimiter*. It hops over numbers and identifiers to the next delimiter, which can be found in variable *dl*. Whether or not

[1] The ALGOL 60 report states that *the sequence of symbols* '**end** <*any sequence not containing* **end** *or* ; *or* **else**>' *is equivalent to* '**end**'.

a number or an identifier was met can be seen from the variable *nflag*: it is set to 1 if a number or an identifier was met, and to 0 otherwise. If *nflag* = 1 the variable *kflag* indicates whether a number (*kflag* = 1) or an identifier (*kflag* = 0) was met. In both cases information indicating what number or identifier was met is given in variable *inw* and, if more information is necessary, in variable *fnw*. In the latter case variable *dflag* is set to 1, otherwise to 0. At most 9 letters (or digits) of an identifier are taken into account. Consequently, identifiers that differ only after the first nine characters are not distinguished. If an identifier consists of less than 5 characters, it can be represented by *inw* alone. In that case the last three bits of *inw* are zero. Note that RND assembles numbers and identifiers from their constituting characters – and does so during both prescan and main scan –, but does no table look–up: all look–up activities are carried out in the main loop of the main scan.

In addition to hopping over identifiers and numbers, RND also hops over the basic symbols '**true**' and '**false**'. These are mapped onto the numbers 0 (for '**true**') and 1 (for '**false**'), i.e., RND delivers into *dl* the code for the delimiter following these symbols and sets *nflag* to 1, *kflag* to 1, *dflag* to 0, and *inw* to 0 or 1.

Although RND, the upper level routine of the hierarchy, is the central interface between lexical scan and the compiler scans, there are a few places in both prescan and main scan where the underlying routine RNS is called. In the first place the contents of strings are skipped (prescan) or read and transferred to the object code (main scan) by calls of RNS. Secondly, in the main scan, comments after an '**end**' symbol are skipped using calls of RNS (during the prescan they are skipped by the main loop thereof using calls of RND). There are two more places in the main scan using RNS: to read the type symbol following the symbol '**own**' in a declaration (for unknown reasons) and to read the symbol following a ']' symbol in a switch designator (for a very specific, technical reason).

Originally, RFS read its characters from an input buffer rather than directly from the paper–tape reader. That buffer was filled by an autonomous process running in parallel with the compiler and driven by the paper–tape reader interrupt. That was a good solution when the reader was slow (about 25 characters/second), but absolutely inadequate for the later installed EL1000 which was capable of reading 1000 characters a second. Recall that the EL X1 executed roughly speaking about 20 instructions in a millisecond, whereas the interrupt handling and buffer administration took about 125 instructions or 6 millisecond per symbol read and delivered (the input buffer being full all the time, retrieving a symbol from the buffer implied a restart of the autonomous reading program, the reading and buffering of a new symbol, and an inactivation of the reading program; in the mean time the interrupt signal was set and before the symbol retrieved from the buffer could be processed the interrupt handler was activated only to find no request for reading). Therefore, we decided to replace the buffer mechanism by a direct access from RFS to the tape reader, leading to a drastic accelleration of the prescan process. Moreover, much attention was given to find further ways to speed up the execution of RFS, using the code table to encode the simple cases in an easy recognizable manner. As a result, the tape was read during the prescan phase at more than half of its maximal speed.

We end this section by a few other remarks on the implementation.

The recognition of word delimiters in NSS is carried out in a rather primitive way. The occurrence of a word delimiter is noticed when an underline symbol '_' is followed by a lower case letter, an 'A' or a 'B'. If that letter happens to be in $\{a, c, d, g, l, o, p, r, u, v, w, B\}$, the identity of the word delimeter is established immediately as '**array**', '**comment**', '**do**', '**goto**', '**label**', '**own**', '**procedure**', '**real**', '**until**', '**value**', '**while**', or '**Boolean**', respectively. Otherwise, a second underlined letter is read. If that is a 't', a third (underlined) letter is read in order to discriminate between '**step**' and '**string**'. Otherwise, if that second letter is in $\{a, e, f, h, r, w\}$ the choice is, given the fact that the first letter was ambiguous, clear: it has to be '**false**', '**begin**', '**if**', '**then**', '**true**', or '**switch**', respectively. Otherwise, the first letter is inspected anew, and given the fact that neither the first letter nor the second was decisive, the choice between '**boolean**', '**end**', '**for**', and '**integer**' is made immediately. After recognition the remainder of the underlined word is skipped. A minor detail is that repetitions of underline symbols are skipped (the underline and the vertical bar are non–advancing symbols on a Flexowriter; therefore, repetitions thereof do not change the print on paper). As stated before, this recognition algorithm is rather primitive and unsafe. For example, '**bagin**' is interpreted as '**false**'! It is, however, also rather fast through the use of a table.

Identifiers are represented by one or two X1 words. If the identifier consists of at most 4 characters, they are stored in *inw*: the last character at bit positions 26 to 21, the second last (if any) at bit position 20 to 15, the third last (if any) at bit positions 14 to 9, and the fourth last (if available) at bit positions 8 to 3. Note that bit positions 2 to 0, the least significant three bits of *inw*, remain zero. If the identifier has more than 4 characters, the fifth character is stored at $inw[21{:}26]$, the fourth at $inw[15{:}20]$, the third at $inw[9{:}14]$, the second at $inw[3{:}8]$, and the first partly at $inw[0{:}2]$, partly in three bits of *fnw* (depending on the number of characters). Since the first character of an identifier is always a letter, letters are internally coded by a value from 10 upto and including 62 (value 36 is unused), and $inw[0{:}2]$ is used for the most significant three bits of the code, these bits are not all zero. In this way a single–word representation can be discriminated from a two–word representation. This is used in the main scan when a name list has to be searched through.

As said before, (unsigned) numbers are assembled by RND. If the number is an integer not exceeding 67108863 (the integer capacity of the El X1), it is represented by one word, *inw*. Otherwise, it is represented by two words, *inw* and *fnw* respectively, as a floating point number in the so–called P9 representation of de X1. The 40 bits mantissa m is scaled between .5 and 1 ($.5 \leq m < 1$). The 26 most significant bits of m are stored in *fnw*, the 14 least significant bits of m together with a sign digit 0 in the head of *inw*. The remaining 12 bits of *inw* are used for the binary exponent e. That exponent should fulfill the requirement $-2048 \leq e \leq +2047$ and the tail of *inw* contains the number $e + 2048$. The conversion from decimal floating to binary floating is carried out in 52 bits precision, with 12 guarding bits. The result is rounded to 40 bits. The conversion uses multiplications or divisions by powers of 10, preferably 10^8, the largest power of 10 represented in the standard X1 system software. In the Pascal version, however, only the first power of 10 is used for reasons of simplicity.

The transformation of the representation of an ALGOL 60 program punched on paper tape in Flexowriter code to a sequence of delimiters possibly separated by constants or identifiers results in an enormous reduction of information. We carried out some measurements on a sample program taken from the PhD thesis of Zonneveld [11]. The text used in these experiments is reproduced in Appendix B. It was typed in ASCII (using ‘ for $\vee$, ˆ for $\wedge$, ˜ for $\neg$, and % for $_{10}$) and transferred to Flexowriter code by means of a Pascal program. We measured:

9198 heptads, of which	1247	shift punchings, dealt with inside RFS
	2730	lay–out punchings, skipped by NSS
	44	punchings of comments skipped by NSS
	2764	one–punching basic symbols
	320	punchings for 160 two–punching symbols
	2092	punchings building 276 word delimiters
	1	lay–out symbol kept in stock by NSS

3200 basic symbols delivered by NSS and stored for reuse in the main scan

1254 delimiters delivered by RND, separated by 658 identifiers, 210 numbers, and 9 logical values

The comment punchings count includes the punchings used for the representation of the symbol ‘**comment**’ itself and of the concluding semicolon. The punching count for word delimiters is inclusive those for ‘**true**’ and ‘**false**’ but exclusive ‘**comment**’. The count of one–punching basic symbols includes 22 lay–out symbols not skipped by NSS because of their occurrence within a string.

Chapter 4

The prescan program

4.1 The art of skipping program text

The main task of the prescan is to construct the prescan list PLI. This list contains, for each block in the ALGOL 60 program, two sublists. The first sublist contains the switch identifiers and the label identifiers declared in the block, the second sublist contains the procedure identifiers declared in the block. These are precisely those identifiers that can be referred to in the block before their declarations. According to the ALGOL 60 report, scalar and array identifiers can also have applied occurrences before their defining occurrence. However, the X1 implementation of ALGOL 60 precribes an order for the declarations of a block: first the scalar variables, next the arrays, and only thereafter the declarations of procedures and switches, in arbitrary order. In these declarations of procedures and switches, the identifiers of all procedures, switches, and labels of the block may be used in applied occurrences.

Only the identifiers are recorded: no other information whatsoever from the declaration is added. It is in the name list (NLI) that is built and manipulated during the main scan that a descriptor is added to each identifier.

Some words must be devoted to what constitutes a block in X1 ALGOL. In the first place, each block in the sense of the ALGOL 60 report constitutes an X1–ALGOL block. Also the declaration of a procedure constitutes a block (containing, e.g., the identifiers of the formal parameters). In addition to these the controlled statement of a for statement constitutes a block. It is this latter mechanism by which a goto statement outside a for statement cannot refer to a label within the for statement, preventing jumps into a for statement. However, some care is taken not to introduce unnecessarily many blocks. If the body of a procedure declaration itself is a block, it is combined with the block containing the identifiers of the formal parameters. If, however, the controlled statement of a for statement is a block in the sense of the ALGOL 60 report, it is treated as a block different from the one that is introduced for the controlled statement itself.

We give a short example. Consider the following ALGOL 60 program:

```
begin integer i;

  procedure p(x); integer x;
  begin switch s:= aa, bb, cc;
    aa:  x:= x - 1;
      goto s[sign(x) + 2];
    bb:
  end;

  procedure q;
  dd:  for i:= 1, 2 do ee:  p(i);

  q;
cc:
  for i:= 1 while i > 0 do
  begin integer i;
    aa:  i:= 0
  end

end
```

This program generates the following PLI:

$[[cc], [q, p], [bb, aa, s], \epsilon, [dd], \epsilon, [ee], \epsilon, \epsilon, \epsilon, [aa], \epsilon]$

In the PLI, blocks are sorted in the same order as the occurrence of their first symbol in the text. Within each sublist, the identifiers occur in retrograde order.

By means of the following two operations the prescan program operates upon the PLI: *fill_prescan_list* and *augment_prescan_list*. The former operation inserts an identifier (stored in *inw* and, perhaps, *fnw*) in some existing sublist, the latter one extends PLI at the end with two new and empty sublists. They use two global variables, *mbc* (for maximum block count) and *bc* (for block count). In *mbc* the number of blocks encountered thusfar is recorded, whilst *bc* gives the number of the current block. Upon block entry *mbc* is incremented by one, the current value of *bc* is saved in a stack, and *bc* is set to *mbc*. Upon block exit *bc* is restored from the stack.

The prescan program itself can best be characterized as 'the art of skipping text'. Its main loop hops, by means of invocations of *read_until_next_delimiter*, from delimiter to delimiter, only paying some attention to it if it is:

- a stringquote open, in order to skip strings;
- '**for**', in order to start a new block in PLI;
- a colon, in order to add the label identifier to PLI;
- '**begin**', in order to see whether it is followed by a declarator (introducing a new block in that case) and to enable a match with the corresponding '**end**';
- '**end**', in order to match it with the corresponding '**begin**' and to check whether it ends a block construction, or perhaps even the program;
- a semicolon, in order to check whether it ends a for statement or a procedure body;
- '**procedure**', in order to add the procedure identifier to PLI and to start a new block in PLI;
- '**switch**', in order to add the switch identifier to the PLI and to skip the switch declaration upto and including its concluding semicolon; or
- '**own**', '**Boolean**', '**integer**', '**real**', '**array**', '**string**', '**label**', or '**value**'. For these symbols the remainder of the corresponding declaration or specification is skipped in an inner loop upto and including its concluding semicolon.

Note that the prescan program never meets a letter, a digit or the symbols '**true**' or '**false**', because these are hopped over by RND (except when occurring within a string).

The main loop as described above can, however, be in one of two states. The current state is recorded in a variable *bflag*. The normal state is $bflag = 0$, whilst $bflag = 1$ indicates the possible processing of specifications. *bflag* is set to 1 whenever the delimiter '**procedure**' is met in the normal state; it is, with some exceptions, reset in each iteration of the main loop. Exceptions occur, for unknown reason, in the iteration following the treatment of a colon, a stringquote open or a '**begin**'.

There are two inner loops. The first one is entered upon the detection of a stringquote open. It skips, by means of invocations of *read_next_symbol*, the contents of the string upto and including the corresponding closing stringquote. Thereafter the next cycle of the main loop is entered without, however, resetting *bflag*.

The other inner loop is used to skip declarations and specifications. It is entered from the main loop after detection or processing one of the delimiters '**own**', '**Boolean**', '**integer**', '**real**', '**array**', '**switch**', '**procedure**', '**string**', '**label**', or '**value**'. It is exitted at the first semicolon, after which the next cycle of the main loop is entered without resetting *bflag*. In this way the parameter list of a procedure, its value list, and its specification lists are skipped by an alternation of a cycle of the main loop and a number of cycles of this inner loop. Inside this inner loop the treatment of the delimiter '**procedure**' is equal to that inside the main loop. In this way the occurrence of a function declaration (starting with '**Boolean**', '**integer**', or '**real**') is properly reacted upon.

The only effect of the state $bflag = 1$ in the main loop is that the delimiters '**switch**' and '**procedure**' are interpreted as specifiers and not as declarators.

Note that array declarations are skipped by an inner loop. In this way the colons that occur in bound pair lists are never taken for a colon marking the occurrence of a label.

(In a later stage we added to the prescan program some code that checks, at each occurrence of a semicolon or '**end**', whether the number of opening parentheses is equal to the number of closing parentheses and whether the number of opening square brackets is the same as the number of closing square brackets met in the text thusfar. In this way a frequently occurring source of troubles could be detected early. The check was carried out during prescan in order to enable the operator to mark the place in the paper tape where the error was detected.)

Because we deal with a context free grammar, a push–down list is needed. It is used to match corresponding '**begin**' and '**end**' symbols and to cater for the block structure of the ALGOL 60 program. Each '**begin**' symbol is pushed onto the stack; it is removed at the occurrence of an '**end**' symbol. If $bflag = 1$, indicating the start of a procedure body, nothing more is added to the stack: it is by this mechanism that a procedure body which is a block by itself does not count as a block in addition to the one that is introduced for the procedure declaration and in which the formal parameters are accomodated. If, on the other hand, $bflag = 0$, and if the '**begin**' symbol is followed by a declarator symbol, indicating the start of a new block, two other values are pushed onto the stack *just below the top–of–stack value (i.e. the* '**begin**' *symbol)*: the current value of bc and the value -1. The latter is used as block marker. The '**begin**' symbol itself continues to be the top–of–stack value.

Also when a '**for**' symbol is encountered, these two values are pushed to the stack too, this time just on top of the stack: bc and -1.

At the occurrence of a semicolon or '**end**' symbol, pairs of a block marker and a saved bc value on top of the stack are popped repeatedly (thereby terminating for statements, which are treated as blocks) until a '**begin**' symbol is found as top–of–stack value. In case of an '**end**' symbol the '**begin**' is popped as well, in case of a semicolon it is preserved in the stack. Each time that a saved bc value is popped in this process it is used to restore variable bc.

For the push operations onto the stack the procedure *fill_t_list* is used (both by the prescan program and the main scan); inspections of the top–of–stack value and pop operations are, however, explicitly coded in the text of the prescan program.

One may wonder whether such a primitive program as the prescan program can properly accomplish its task, the construction of the prescan list, for any syntactically correct ALGOL 60 program, and in fact it does not. We found a number of flaws but in practice they hardly mattered: most programmers don't write grammatically complex programs. I remember only one user problem that we could trace back to a shortcoming of the prescan program, and it was easily circumvented.

A first mistake, rather unimportant, is the way in which comments between an '**end**' symbol and the subsequent semicolon, '**else**' or '**end**' symbol are dealt with. The comment symbols are skipped by the main cycle of the prescan program and consequently there is a reaction upon the occurrence of those symbols the prescan program is interested in. In the X1 ALGOL 60 user's manual the occurrence of the symbols '**begin**', '**comment**' and of stringquotes are explicitly forbidden, but also symbols like '**for**' and '**procedure**' better do not occur in these contexts, as is illustrated by the following example:

```
begin integer i;
  for i:= 0 do
  begin AA: end for i, BB: ;
CC:
end
```

producing:

$[[CC], \epsilon, [AA], \epsilon, [BB], \epsilon]$

as prescan list in stead of:

$[[CC], \epsilon, [AA], \epsilon]$

We notice already here that in the main scan program there is a separate loop of only 6 X1 instructions for skipping this kind of comments neatly, and it is incomprehensible why the same solution is not used in the prescan program. Then no exclusion rule would have been necessary in the user manual, and the prescan program and the main scan program would have had the same treatment of comments. The true solution would have been to skip such comments already in the lexical scan part of the compiler: that's where it belongs!

A more serious flaw is caused by the way the block structure is treated. For an ALGOL 60 block '**begin** <declarations> <statements> **end**' the block marker -1 in the stack is not removed upon reading of the '**end**' symbol, but only at the next semicolon or '**end**' (at the same level). Consequently, for the following ALGOL 60 program:

```
begin
  if 0 < 1
  then AA: begin integer i;
              BB:
           end
  else CC: begin integer i;
              DD:
           end
end
```

the prescan program generates the following prescan list:

$[[AA], \epsilon, [CC, BB], \epsilon, [DD], \epsilon]$

instead of:

$[[CC, AA], \epsilon, [BB], \epsilon, [DD], \epsilon]$

The faulty prescan list leads to an endless loop within the main scan.

4.2 Representation of the prescan list

During prescan and main scan the working space of the latest version of the X1 ALGOL 60 compiler ran from store address 1933 (1–28–13)[1] upto 6783 (6–19–31). In this space all lists had to be accomodated with the exception of the compiler stack and of the outputbuffer for the console typewriter. For the former 128 words were reserved from store address 800 (0–25–0) upto 927 (0–28–31).

The execution of the prescan program generates two lists: the prescan list PLI and a coding of the input text as produced by *NSS*, packed 3 symbols in a word. The text words were stored from address 1941 onwards, PLI was build at the end of the available space; its last word had address 6783.

The representation of PLI was just a linked list. The words coding the identifiers of a sublist of PLI were written one after another without any separation. Each sublist, however was preceded by a link refering to (the link preceding) the next sublist. All these

[1] In the X1–practice it was customary to denote addresses in the number system with base 32: a 15–bit address is then split into three 5–bit parts. For the Mathematical Centre X1 addresses ran from 0–0–0 upto 11–31–31 and, for the read–only part of the store, from 24–0–0 onwards.

links were forward references. After the last sublist a backward reference was included as an endmarker. PLI is initialized as:

address	contents
6781	6782
6782	6783
6783	6782

representing the two (as yet) empty sublists of the outermost block.

The prescan list for the first example read:

$$[[cc], [q, p], [bb, aa, s], \epsilon, [dd], \epsilon, [ee], \epsilon, \epsilon, \epsilon, [aa], \epsilon].$$

Its representation is given in Figure 2.

A consequence of this representation is that the insertion of an identifier in one of the sublists, or of two new (empty) sublists at the end of PLI is quite a complex operation: shifting part of the list downwards in order to create one or two empty places and updating the links in the lower part of the chain. In order to keep the amount of shifting as small as possible identifiers are inserted at the front end of the appropriate sublist.

The chosen representation is, on the other hand, quite fit for use during the main scan.

4.3 Quantitative aspects

In order to get an impression of the efficiency of the prescan program we carried out some measurements on the sample program of Zonneveld that was also used in the previous chapter. What we could easily measure was the number of instructions executed between two successive read instructions (the count including one of these).

The paper–tape reader of the X1 was able to read 1000 punchings a second. The minimal time between two successive reads was therefore 1 millisecond. Taking as average instruction time about 50 microsecond, the X1 was capable of executing some 20 instructions per millisecond. If less than 20 instructions were executed between two read instructions, the X1 had to wait until the next read result was available, whilst the execution of more than 20 instructions between two read instructions lead to an activation of the brakes and a slow–down of the tape reader.

From our measurements we calculated the average number of instructions executed between reads, replacing any count less than 20 by 20. The resulting average was 33.8

address	contents	comments
6762	6764	
6763	25559040	cc
6764	6767	
6765	54525952	q
6766	52428800	p
6767	6771	
6768	23429120	bb
6769	21299200	aa
6770	58720256	s
6771	6772	ε
6772	6774	
6773	27688960	dd
6774	6775	ε
6775	6777	
6776	29818880	ee
6777	6778	ε
6778	6779	ε
6779	6780	ε
6780	6782	
6781	21299200	aa
6782	6783	ε
6783	6782	

Figure 2: store representation of $[[cc], [q, p], [bb, aa, s], \epsilon, [dd], \epsilon, [ee], \epsilon, \epsilon, \epsilon, [aa], \epsilon]$

instructions, suggesting that the tape reader would have run at 60% of its maximum speed for this program.

A detailed analysis of the available 9198 number–of–instructions–between–successive–reads can be given. We want to relate these figures to specific activities in the layers of the lexical scan and in the prescan program itself. Before doing so we tried to eliminate the effects of two different sources of a kind of noise. In the first place the second level of the lexical routines, *NSS*, reads at some occasions one Flexowriter symbol in advance, which then is stored in an internal buffer. At the next invocation of *NSS* this symbol is taken from that buffer instead of reading it from the tape reader. In the second place, within the third level of the lexical routines, *RNS*, the symbol obtained from *NSS* is stored in the text buffer. This takes 11 instructions but at one of each three invocations an additional 7 instructions are executed for starting a new text–buffer word (remember that in the text buffer 3 symbols are stored per word).

A first observation is that the 9198 heptades read by the tape reader lead to only 3200 symbols delivered by *NSS*. This means that 5998 heptades are 'absorbed' by *RFS* and *NSS*. In 5145 of the cases the number of instructions executed between two successive reads is 20 or less, and in 6007 cases 27 or less. With some exceptions these will correspond to absorbed heptades, and the average number of instructions between reads is for these cases only 20.4, replacing again numbers less than 20 by 20.

For a second observation we mention that the PLI produced for this program counts 36 blocks (of which 32 for blocks without any identifier) and 8 short (i.e. one–word) identifiers, resulting in a total PLI length of 81 words. The first block introduction (for procedure f) takes 163 instructions, including insertion of its identifier, whilst the insertion of label identifier A requires 114 instructions, the incorporation of the first for block 181 instructions, and that of the last for block 744! We see here clearly the effect of the steadily increasing amount of work for adding new sublists at the end of PLI (all numbers mentioned here are the number of instructions between successive reads). Block introduction or name insertion costs, on the average, 413.0 instructions. As a result the tape reader halted noticeably at the occurrence of labels, switch– or procedure identifiers and '**for**' symbols.

For the remaining symbols we measured an average number of instructions between successive reads of 54.6. This caters for the activities at all the levels of the lexical scan and at the prescan level itself. The lowest number above 27 that occurs is 34, the biggest one not related to PLI increments is 133. Typical numbers are 36 and 43 for the letters and digits of an identifier and 50 and 57 for the digits of a number (here we should mention that for each digit of a number two multiplications are executed with an execution time of 500 μsec each. Therefore the figures 50 and 57 could also be read as 68 or 75).

The prescan as a whole takes 292 810 instructions, 256 848 (88%) of which are spent in the lexical scan. In more detail, RFS requires the execution of 95 722, NSS of 63 990, RNS of 42 684 and RND of 54 452 instructions.

Chapter 5

Main scan

During the main scan the object program is generated. In the original version of the compiler the source text was read from paper tape a second time and the object program was punched, also on paper tape. The latter was a rather time–consuming process as the tape punch ran at a speed of 25 punchings a second. In the latest load–and–go version of the compiler the source text was taken from store and the object program as produced during the main scan was stored in compressed form. After completion of the main scan it was decompressed and stored at its final place by the loading phase of the compiler.

Inputs to the main scan are the source text (as stored by RNS during prescan) and the prescan list PLI. Outputs are: the list of object instructions RLI (in its compressed form), the list of future references FLI, the list of constants KLI and some numbers: the lengths of RLI, of FLI, and of KLI and GVC, the first free address of the execution stack.

The structure of the main scan resembles that of the prescan program. There is one central loop and some inner loops (for dealing with special constructs like strings, formal parameter lists etc.). The central loop starts with a call of RND, whereafter there is a case construct on the delimiter just read. In contrast to the prescan program, the main scan has a separate case for almost every delimiter, in which a piece of object program is generated and the appropriate administrative actions are carried out. There is also a state, to control the activities in the central loop, and a push–down list to cater for the context–free character of the ALGOL 60 grammar.

The state – in the prescan program just one boolean – is much more extended than during prescan and is used to record the context. Also the stack is used for many more purposes than in the prescan program.

5.1 Structure of the object program

The object program generated by the X1 ALGOL 60 compiler has been documented by Dijkstra [5]. This report is in Dutch and presents a mixture of a description of the object program itself and of its working during execution.

The compiled program is in terms of 101 operators that are coded in 'The Complex', a complex of run–time subroutines. Many of the operators have parameters, which are transferred to the subroutine in one of the X1 registers. Therefore, the object program is full of instructions to load a parameter value, e.g. the address of a variable, into a register, followed by a call to one of the subroutines of the complex. A full list of the operators is given in Appendix C. The complex had two versions: ALD, using a 54 bits representation for real numbers, and ALS, using 27 bits to represent these. The latter ran slightly faster and used less space for storing real arrays but was hardly used in practice.

The object program is transferred by the main scan to its destination (paper tape originally, store later) by means of the procedure *fill_result_list* (FRL). FRL has two parameters:

- the OPC–value, which is either the number of an operator from the complex ($8 \leq OPC \leq 109$), or has one of the values 0, 1, 2, or 3.
- a word w. For an operator from the complex the value of w is irrelevant, otherwise it is an X1 instruction (or in a few cases a constant or a code word), to be incorporated into the object program. The OPC–value then indicates whether in the loading phase following the main scan the instruction should be taken as it is (OPC = 0), the begin address of the object program should be added to it (OPC = 1), the address part of the instruction should be replaced by the corresponding entry in the future list (a list of future references produced by the main scan) (OPC = 2), or the begin address of the constant list should be added to the instruction (OPC = 3).

The result list RLI (both its version on paper tape and the one stored in computer store) is just an encoding of these parameters. It is only in the loading phase of the ALGOL 60 system that OPC–values ≥ 8 are replaced by the corresponding subroutine calls of the complex and the meaning of OPC–values ≤ 3 for w–values is taken into account.

Apart from parameter–loading instructions, mainly jump instructions occur as explicit instructions, either coded with OPC = 1 (for backward jumps) or with OPC = 2 (for forward jumps). Only 11 instruction types (with different opcodes) do occur as explicit instructions.

The address of a variable can be either 'static' (for variables declared in the outermost block) or 'dynamic' (for variables declared in inner blocks). A dynamic address consists of two parts, a block number n and a displacement d relative to the begin of the block cell in the execution stack. As a parameter to an operator it is coded as $32 * d + n$.

As an example we present in Figure 3 the piece of object program produced as the compilation of the statement '`i:= 2 + i * i`', assuming the dynamic address $n = 1, d = 7$ for i and relative position 29 for constant 2 in the constant list.

OPC	w	explanation
0	2S 225 A	load dynamic address of `i` in register S
16		TIAD, take integer address dynamic
3	2B 29 A	load static address of 2 in register B
34		TIRS, take integer result static
0	2S 225 A	load dynamic address of `i` in register S
33		TIRD, take integer result dynamic
0	2S 225 A	load dynamic address of `i` in register S
48		MUID, multiply integer dynamic
59		ADD, add
85		ST, store

Figure 3: object code for the statement '`i:= 2 + i * i`'

The translation is syntax directed and in polish–reversed form. The load instructions are generated on the basis of the identifier or constant assembled by RND, the operations are formed on the basis of the delimiters and kept in the compiler stack until they can be inserted in the object program. The latter is regulated by the priorities of operator and context. The assignment symbol '`:=`' is considered an operator with lowest priority. Where possible an operator is combined with a preceding take. MUID is such a combination of TIRD and MUL (multiply). All of the instructions of the example given above are generated inside procedure *production_of_object_program*. We come back to these points in a separate section.

The translation is such that the code corresponding to applied occurrences of identifiers is generated during the analysis of the delimiter immediately following it. There is one exception to this rule: the code for a switch identifier in a switch designator is generated after the code for the index expression. In this case the identity of the switch designator has to be saved in the stack (during analysis of '[') and to be popped later (during analysis of ']').

As further examples we present in Figure 4 the translation of a procedure statement

	OPC	w	explanation
511:	1	2B 18 A	load begin address of `SUM` in register B
	2	2T 3	jump over translation of parameters
513:	3	2B 0 A	load static address of x in register B
	15		TRAS, take real address static
	0	2B 138 A	load static address of `i` in register B
	34		TIRS, take integer result static
	56		IND, indexer
	13		EIS, end of implicit subroutine
	1	0A 513 C	codeword for parameter `x[i]`
	3	0A 5 A	codeword for parameter `10`
	3	0A 4 A	codeword for parameter `1`
	0	0A 138 A	codeword for parameter `i`
523:	0	2A 4 A	load number of parameters in register A
	9		ETMP, extransmark procedure

Figure 4: object code for the statement '`SUM(i,1,10,x[i])`'

'`SUM(i,1,10,x[i])`' and in Figure 5 that of a goto statement '**goto** `s[i]`'.

OPC	w	explanation
0	2B 138 A	load static address of `i` in register B
34		TIRS, take integer result static
29		SSI, store switch index (in location 48)
1	2T 65 A	jump to code for declaration of `s`

Figure 5: object code for the statement '**goto** `s[i]`'

Here we supposed that:

- the translation of the procedure statement starts at (relative) address 511 of result list RLI;
- the translation of the declaration of `SUM` starts at (relative) address 18 of result list RLI;
- the contents of word 3 of future list FLI contains the (relative) address of the first instruction following the translation of the actual parameters, i.e., 523;
- the storage function[1] of array `x` is located from word 0 of constant list KLI;

[1]The 'storage function' of an array is a number of words containing the information nescessary to

- the static address of variable `i` is 138;
- constants `10` and `1` are located in words 5 and 4 of constant list KLI;
- the (relative) address of the entry point for the code for the (switch) declaration of `s` is 65.

The first instruction of the implicit subroutine for parameter `x[i]` is located at (relative) address 513 in result list RLI;

The subroutine ETMP in the complex gets in fact 3 parameters: the address of the code for `SUM` in register B, the number of actual parameters in register A, and the subroutine link (so that it, a.o., can find the code words describing the actual parameters). Note that for simple actual parameters like variables and constants one parameter code word suffices to encode them. For each more complex actual parameter a piece of code, called implicit subroutine, is generated preceding the code words. In that case the code word contains, a.o., the (relative) address of that piece of object code.

5.2 The execution model

Although it is not very relevant for the discussion of the main scan of the compiler, we will nevertheless present some information about the execution model.

5.2.1 The execution stack

The main data structure during execution is the execution stack. It is a list of block cells, one for each block in execution, in order of the moment of block entry. Apart from the first cell — that for the outermost block — each cell has the same structure: 5 words of link data, the locations of the formal parameters (2 words per parameter), the locations for local scalar variables (1 word for integers and booleans, 2 words for reals), the storage functions of local arrays ((3 + the array dimension) words), the storage functions of value arrays (8 words per array), space for the elements of local and value arrays (again 1 word per element for arrays of type integer or boolean, 2 words per element for real arrays), and the expression stack. The begin address of the block cell is called *pp* (procedure pointer), the begin address of the expression stack *wp* (working space pointer), and the address of the first location following the expression stack *ap* (accumulator pointer). There are global variables AP, WP, PP, and BN containing these pointers and the block number of the blockcell currently in execution.

---

compute the address of an array element given the index values.

The link data consist of:

- the pp–value of the most recent incarnation of the lexicographically enclosing block (the 'static' link),
- the wp–value, the pp–value and the block number just before block entrance (the 'dynamic' link), and
- the return address (the subroutine link proper).

These values are written by the complex subroutine ETMP, for a procedure statement of a non–formal procedure, or by ETMR (extransmark result, OPC 8), for a function designator of a non–formal type–procedure. These subroutines also reserve (and prepare the contents of) the locations of the formal parameters on the basis of the code words just preceding the call of ETMP or ETMR.

The locations for the formal parameters contain:

- for a parameter called by name and for an array parameter called by value: a two–location code word characterizing the corresponding actual parameter, which can be interpreted by OPC 18 (TFA, take formal address) or OPC 35 (TFR, take formal result),
- for all other parameters called by value: their value in one (for integer or Boolean values) or two (for real values) words.

ETMP and ETMR prepare the locations for all formal parameters as if they are called by name; the transformation of the code words to values for value parameters is carried out by (the object code of) the procedure declaration itself.

The locations for all simple local variables together are reserved by three instructions in the code for the procedure declaration, simply incrementing AP and WP by the same amount. These instructions are generated by the compiler procedure *reservation_of_local_variables* which is called when a type declaration is not followed by another type declaration. In this context it is important that all type declarations of a block are grouped together and precede all other declarations of the block.

After the reservation of the locations for all simple local variables the storage functions for the local arrays are constructed, thereby incrementing the values of AP and WP anew by (array dimension + 3) per local array. The complex routines RSF (real arrays storage function frame, OPC 90) and ISF (integer arrays storage function frame, OPC 91) play a role here. Only after completion of the construction of the storage functions for all local arrays the storage functions for the value arrays are constructed using the formal parameter code words build by ETMP or ETMR, and incrementing the values of AP and WP by 8 per value array (this restricts the dimension of arrays called by value to at most

5). This work is done in the compex routines RVA (real value array storage function frame, OPC 92) or IVA (integer value array storage function frame, OPC 93). Thereupon the space for the elements of all local and value arrays is reserved using LAP (local array positioning, OPC 94) or VAP (value array positioning, OPC 95). Both LAP and VAP increment AP and WP. VAP is also responsible for making the copy of the elements of the actual parameter array. The amount of space required for the array elements is not known at compile time, and does not play any role in the dynamic address system (the displacement part of which being restricted to at most 1023). RVA, IVA, LAP, and VAP are generated by the compiler procedure *reservation_of_arrays* which is called at the occurrence of the first delimiter implying that no more declarations of local arrays follow. Here it is important that all array declarations of a block precede the declarations of switches and procedures.

The expression stack consists of 4–word cells. The last word of each cells specifies its type (-1 for real values, -0 for integer values, some value $\geq +0$ for addresses). Integer values and addresses are given by the first word of a cell, real values use the first three cell words (a mantissa of 52 bits + 2 sign bits, a binary exponent of 27 bits).

ETMR reserves a 4–word cell on top of the expression stack before constructing the new block cell. Moreover, both ETMP and ETMR fill one word just below the new block cell, ETMP giving it the value -0 and ETMR the address of the 4–word cell for the result. This word is inspected by OPC 87 (STP, store procedure value) to see whether the calling environment of a type procedure needs the result or not, and if so, where it should be stored.

The translation of the procedure declaration:

```
real procedure SUM(i,a,b,ti);
value b; integer i,a,b; real ti;
begin real s;
  s:= 0;
  for i:= a step 1 until b do s:= s + ti;
  SUM:= s
end;
```

is given in Figure 6.

In Figure 6 we assumed that location 24 of the future list contains the (relative) address of the instruction following the return instruction, that location 8 of the constant list contains constant 0, and that procedure `SUM` is declared in the outermost block.

OPC	w	explanation
2	2T 24	jump over procedure declaration
0	2B 1 A	load block number in register B
89		SCC, short circuit
0	2S 41 A	load dynamic address of `b` in register S
16		TIAD, take integer address dynamic
0	2S 41 A	load dynamic address of `b` in register S
35		TFR, take formal result
85		ST, store
0	2A 2 A	load length local area in register A
0	4A 49	increment WP
0	4A 50	increment AP
0	2S 45 A	load dynamic address of `s` in register S
14		TRAD, take real address dynamic
3	2B 8 A	load static address of 0 in register B
34		TIRS, take integer result static
85		ST, store
...	...	translation of the for statement
0	2S 45 A	load dynamic address of `s` in register S
31		TRRD, take real result dynamic
0	2B 0 A	load block nr of enclosing block in register B
87		STP, store procedure value
12		RET, return

Figure 6: object code for the declaration of real procedure '`SUM`'

5.2.2 The display

A second data structure that plays an important role in the execution phase of an ALGOL 60 program is the display. It is a list *disp* of length BN + 1, and its elements are the PP–values of the static chain. More precisely, $disp[0] = 0$, $disp[\mathrm{BN}] = \mathrm{PP}$, whereas for all i, $1 \leq i < \mathrm{BN}$, we have $disp[i]$ = the static link from the block cell starting at $disp[i+1]$.

disp is used for converting dynamic addresses to static addresses: the static address corresponding to the dynamic address $32 * d + n$ is $disp[n] + d$.

disp is updated during block entrance (by routine SCC, short circuit, OPC 89, from the complex), block exit (by routine RET, return, OPC 12), just before a jump that leads to a label outside the block currently in execution (by GTA, goto adjustment, OPC 28) and at the start and at the end of the execution of an implicit subroutine (the translation of a non–trivial actual parameter).

5.3 The context state

As stated before, the structure of the main–scan program is a loop in which a call of RND (read_until_next_delimiter) is followed by a case analysis with respect to the delimiter just read. The interpretation of that delimiter often depends on the context, which is kept in a number of state variables, the context state.

The context state can be described by the 6–tuple:

$$(\mathit{eflag}, \mathit{oflag}, \mathit{mflag}, \mathit{iflag}, \mathit{vflag}, \mathit{sflag})$$

These flags are boolean variables (coded 0 for '**false**' and 1 for '**true**'), and have the following meaning:

flag	the context is
eflag	an expression
oflag	the start of an expression
mflag	an actual parameter list
iflag	a subscript list
vflag	a for clause
sflag	a switch declaration

eflag and *oflag* are set after the delimiters '**if**', '**do**', '`:=`', '`(`', '`[`', and '**array**'. There are several places where *eflag* is reacted upon, e.g. to determine whether a procedure call is a procedure statement or a function designator. *oflag* is reacted upon at one place only; it determines whether the delimiters '`+`' and '`-`' should be interpreted as binary ($\mathit{oflag} = 0$) or unary ($\mathit{oflag} = 1$) operators. It is reset in each call of RND.

mflag is set after a procedure identifier followed by '`(`', after pushing its old value to the stack. It is reacted upon in the analysis of the delimiters '`,`' and '`)`', interpreting them in case of $\mathit{mflag} = 1$ as actual parameter list separator and actual parameter list closing parenthesis, respectively. Its old value is popped when dealing with the latter (after generating the parameter code words and the ETMP or FTMP instruction). Moreover, *mflag* is reset at the beginning of expressions between parentheses and of subscript lists after pushing its old value, which is popped at the occurrence of the corresponding closing bracket. (Note: *mflag* does not play any role in procedure declarations, since the procedure heading is analysed in an inner loop of the case '**procedure**' of the central loop).

iflag is set after reading the delimiter '`[`'. It is reacted upon in the analysis of the delimiter '`,`', interpreting that delimiter as a separator in a subscript list or a bound pair list. Its old value is temporarily saved in the stack during the scan of an actual parameter list, a subscript list, and a bound pair list. In the first case it is also reset.

vflag is set after reading the delimiter '**for**' and reset after reading '**do**'. It is reacted upon in the analysis of the delimiters '`:=`' and '`,`', interpreting these as delimiters in a for clause. During the scan of an actual parameter list *vflag* is reset, but its old value is kept in the stack.

sflag is set after reading the delimiter '**switch**'. It is reacted upon in the analysis of the delimiters '`:=`' and '`,`', interpreting these as delimiters in a switch declaration. It is reset at the end of the switch declaration, when meeting a semicolon when $sflag = 1$.

In case of the opening of a new context part of the old context state is saved in the stack and retrieved from the stack at return to the old context. This is carried out in an ad hoc fashion: each case only that part of the state is pushed that is relevant after the return from the new context and that (possibly) has an other value in the new context. As an example we mention that *vflag* is saved in the stack in the analysis of the delimiter '`(`' provided that it is the opening parenthesis of an actual parameter list (this changes the interpretation of commas).

There are three more flags: *pflag*, *jflag*, and *fflag*. They play a role in the interpretation of identifiers and mainly affect the generation of the object program. They are reset in each call of RND. If RND hopped over an identifier (setting $nflag = 1$ and $kflag = 0$), the code of the central loop following the call of RND will set, according to the data stored in the namelist, *pflag* if the identifier is the name of a procedure, *jflag* if it is the name of a label or switch, and *fflag* if it is the name of a formal parameter. Moreover, *jflag* is set at the begin of a switch declaration. As said, these flags mainly affect the generation of the object program. In some special situations they influence also the interpretation of a delimiter. An example of the latter is the interpretation of the delimiter '`(`': if $pflag = 1$ it is interpreted as the opening parenthesis of an actual parameter list. At one occasion *jflag* is even pushed to the stack: at the occurrence of the delimiter '`[`' its value is saved in the stack and retrieved from it at the occurrence of '`]`'. Also *fflag* is pushed at '`(`' and '`[`' and popped at the corresponding closing parentheses. In these cases the information about the identifier concerned is needed at a later stage. Since the values of these three flags are determined anew at each delimiter of the text, we do not consider them part of the context state.

5.4 The name list

During the main scan the compiler maintains a symbol table called the name list NLI. It contains all identifiers that are in scope. There is no block structure visible in this list: the only structure present is the list's ordering: the identifiers of the most recently entered block structure are at the end of the list. Searches scan NLI in backward order: the search for an identifier starts at the end of NLI and continues until the identifier is found or the begin of the list is reached (in which case the compilation is halted with error stop 7: "undeclared identifier").

In contrast to the prescan list PLI, NLI contains for each identifier a one–word descriptor (immediately following the one– or two–word coding of the identifier). The interpretation of the 27 bits of this descriptor is depicted in Figure 7.[2]

Note that array identifiers are not marked as such in their descriptor.

At the begin of a new block in the text the current length of NLI (recorded in the compiler variable *nlsc*) is saved in the stack. Thereupon the next two sublists of the prescan list PLI are moved to (the end of) NLI, adding to each identifier a descriptor: $d17 + d15 + d19 * bn$ for the label and switch identifiers of the first sublist, $d18+d15+d19*bn$ for the procedure identifiers of the second sublist, where bn is the block's blocknumber.

In the case of a procedure declaration the formal parameter list is scanned after this augmentation of the name list. The formal parameter identifiers are added to the name list with a descriptor containing their dynamic address and the bits $d16 + d15$. Thereupon the value list is scanned, the identifiers are searched for and $d26$ is added to their descriptors. Next the specifications are scanned, adding $d17$ for identifiers specified '**label**' or '**switch**', $d18$ for identifiers specified {<type>} '**procedure**', and $d19$ for identifiers specified '**integer**' or '**Boolean**' {'**array**'}. Moreover, for identifiers specified '**real**', '**integer**' or '**Boolean**' that occurred in the value list (having $d26 = 1$) $d26$ is reset and code is generated for evaluating the corresponding actual parameter and storing its value at the location reserved for the actual parameter code word.

According to the restrictions for X1 ALGOL 60 programs all formal parameters should be specified. This plays a role in the code to be generated for a statement like '`p(s[i])`' where '`s`' is a formal identifier. For a formal switch identifier this code differs from the code for a formal array identifier.

At each applied occurrence of a nonformal label, switch, or procedure identifier the com-

[2] In the X1 tradition the 27 bits of a word were denoted by $d26, d25, \ldots, d0$, $d26$ being the most significant bit.

bits	interpretation
$d26$	1 for a formal value parameter for which not yet its evaluation code has been generated, 0 otherwise
$d25, d24$	OPC–value for a nonformal label, switch or procedure identifier
$d23 \cdots d19$	for a nonformal label, switch, or procedure identifier: its block number, otherwise $d23 \cdots d20$ all 0
$d19$	1 for an integer or Boolean type variable or array or for a formal parameter occurring in the value list and specified integer or Boolean (array)
$d18$	1 for a formal or nonformal procedure identifier
$d17$	1 for a formal or nonformal label or switch identifier
$d16$	1 for a formal name parameter identifier
$d15$	for a nonformal label, switch, or procedure identifier: 1 before its first occurrence in the text, 0 thereafter for a simple variable, an array, or a formal parameter: 1 if it has a dynamic execution–stack address, 0 otherwise
$d14 \cdots d0$	object–code address (for a label, switch, or procedure), future–list location (idem), or execution–stack address (for a simple variable, array, or formal parameter)

Figure 7: The 27 bits of a descriptor in the name list

piler routine *test_first_occurrence* is called. If $d15 = 1$, i.e., if it is its first texual occurrence (which therefore precedes its defining occurrence), $d15$ is reset, $d25$ is set (giving the identifier an OPC value of 2), a place in the future list is reserved for the as yet unknown object–code address, and the address of that place is filled in in bits $d14 \cdots d0$ of the descriptor.

At the defining occurrence of a label, switch, or procedure identifier the compiler routine *label_declaration* is invocated. If $d15 = 1$, $d15$ is reset, $d24$ is set (giving the identifier an OPC value of 1), and the current length of the object code (recorded in the compiler variable *rlsc*) is filled in in bits $d14 \cdots d0$ of the descriptor. If, on the other hand, $d15 = 0$, the value of *rlsc* is stored in the future list at the location stored in bits $d14 \cdots d0$ of the descriptor. Note that in that case all applied occurrences of that identifier are addressed with an OPC value of 2 and a reference to the future list, also those following its defining occurrence. Another task of *label_declaration* is the output to the console typewriter of the label, switch, or procedure identifier, followed by its (relative) object–code address in 32–ary scale notation.

All other defining occurrences of identifiers (i.e. of scalar variables and of arrays) lead to the addition of that identifier at the end of the name list. They get static addresses if their declarations occur in the outermost block, otherwise they get dynamic addresses[3]. Therefore, they get a descriptor with $d14 \cdots d0$ filled in, $d15 = 1$ in case of dynamic addressing, and $d19 = 1$ in case of an integer or Boolean (array) type.

At the end of a block the old length of NLI is retrieved from the stack and stored in *nlsc*, thereby effectively removing all local identifiers of the block from the name list.

Identifiers are searched for in the name list by the compiler routine *look_for_name*. If the identifier is found then its descriptor is copied to the compiler variable *id*; the (relative) position of the descriptor within the name list is stored in another compiler variable *nid*. Note that *id* and *nid* potentially change value after each call of *read_until_next_delimiter* in the main cycle. In general, the old values need not to be saved in the stack. In four cases, however, *nid* is pushed to the stack: during the scan of the <switch list> of a <switch declaration>, of the <subscript expression> of a <switch designator>, of the <bound pair list> of an <array declaration>, and of the <subscript list> of a <subscripted variable>. In the first two cases the old value is used afterwards indeed.

At the start of the main scan the name list is prefilled with a number of identifiers. These are the identifiers of those procedures and functions that are available without declaration. To these belong the standard functions `abs`, `sign`, `sqrt`, `sin`, `cos`, `ln`, `exp`, `entier`, and `arctan`, some input and output procedures as `read`, `print`, `TAB`, `SPACE`, `PRINTTEXT`, `FLOT`, `FIXT`, and `ABSFIXT`, and some frequently used functions. They fall apart in two catagories:

- The first *nlscop* words of this prefill belong to procedures that are treated as operators. A call of such a procedure is translated by code transferring its parameter values to the stack, followed by an invocation of the corresponding routine from the complex of runtime routines.
- The remaining $nlsc0 - nlscop$ words belong to procedures that in the loading phase of the system are selectively added to the code from some library source.

For the procedures of the first category the discriptor has the value $d18 + 12 * 256 + n + o$,

[3]Own scalar variables and own arrays always get static addresses, as if they were declared in the outermost block.

where o is the OPC–value of the operator and either $76 \leq o \leq 84$ and $n = 57$ or $102 \leq o \leq 108$ and $n = 40$. It is likely that the latter group is from a historically later period than the first group. OPC–value 81 was reserved for the function `arctan` which later was moved to the library, using a different algorithm.

For the procedures of the second category the descriptor has the value $d18 + d15 + m$, where m is the number of the routine in the library.

Finally we remark that clearly the name list has been designed to occupy a minimal amount of core store. The identifiers of blocks that have been scanned completely do not require store any longer, where as for blocks in the yet unscanned part of the text only the local label, switch, and procedure identifiers are kept in the prescan list.

5.5 The constant list

The constant list KLI is built during the main scan and contains all constants that occurred in the program thusfar, including values for the logical values[4] '**true**' and '**false**'. Each constant met by RND in the central loop of the main scan is searched for by the compiler routine *look_for_constant*. If the constant is not found it is added to the list. Moreover, *look_for_constant* assigns the value $d25 + d24 +$ the (relative) address of the constant in KLI + (if $dflag = 0$ then $d19$ else 0) to the compiler variable *id* as a pseudo descriptor.

Another contribution to KLI are the storage functions of the arrays declared in the outermost block and of all own arrays (which, in fact, are treated as if they were declared in the outermost block). The storage function consists of (3 + array dimension) numbers which are computed by the compiler and stored in KLI by means of the compiler routine *fill_constant_list*. According to the restrictions for X1 ALGOL 60 programs the array bounds for these arrays should be numbers (instead of <arithmetic expression>s). The array bound numbers are read by a separate inner loop of the main scan program and not added to the constant list but processed immediately in the construction of the storage function.

[4] ALGOL 60 terminology.

5.6 The future list

In the generation of the object program it occurs frequently that an instruction refers to an address in the object program that is not yet known. This is the case for applied occurrences of label, switch and procedure identifiers that precede their defining occurrences, but also for the forward jumps that are used in the code for certain ALGOL program constructs like conditional statements and for statements. In such a case the first free location of the future list FLI is reserved for the yet unknown address and the index of that location is used as address in the instruction (marked as such by an OPC value of 2). During the loading phase of the ALGOL system the future list references in the object program are replaced by the contents of the future list.

In the case of a forward reference to the declaration of an identifier the index of the reserved location in FLI is stored in the descriptor of that identifier in the name list as described in a previous section. In the case of a forward reference in a program construct the index of the reserved location is saved in the stack or in a global compiler variable and retrieved when the object–code address concerned is defined. Then the latter can be filled in in the FLI location.

As an example of the use of FLI we give the translation of the expression '`(`**if** `x > y` **then** `x` **else** `y)`' in Figure 8. Suppose that preceding the compilation of that expression the length of the object code *rlsc* = 182 and the length of the future list *flsc* = 18. Let, moreover, both `x` and `y` be of type real and statically addressed with 138 and 140, respectively.

After generating this code, FLI[18] = 192, FLI[19] = 194, *flsc* = 20 and *rlsc* = 194.

The future list is also used when one of the library routines is used in the program. At the first occurrence of its identifier a location is reserved in FLI which is filled by the descriptor of that identifier. In the name list itself that descriptor is replaced by $d18 + d24 + d25 +$ the index of the reserved location in FLI. These actions are carried out in the compiler procedure *test_first_occurrence*.

The reservation of a location in the future list is done explicitely by reading the value of *flsc* and incrementing it. The assignment of a value to such a location is always carried out by means of the compiler procedure *fill_future_list*, which takes two parameters: the (absolute) store address of the location and the value to be assigned. As we will discuss elsewhere it may be nescessary to enlarge the area reserved for the future list first before the assignment can be effectuated, but this is encapsulated totally inside *fill_future_list*.

	OPC	w	explanation
182:	0	2B 138 A	load static address of `x` in register B
	32		TRRS, take real result static
	0	2B 140 A	load static address of `y` in register B
	32		TRRS, take real result static
	65		MOR, more
	30		CAC, copy Boolean accumulator into condition
	2	N 2T 18	if condition = NO, jump to else part
	0	2B 138 A	load static address of `x` in register B
	32		TRRS, take real result static
	2	2T 19	jump over else part
192:	0	2B 140 A	load static address of `y` in register B
	32		TRRS, take real result static
194:			

Figure 8: The translation of the expression ‘(**if** `x > y` **then** `x` **else** `y`)’

5.7 The translation of a for statement

The translation of a <for statement> contains many forward references for which the future list is heavily used again. A scheme for the translation of ‘**for** <variable> `:=` <for list> **do** <statement>’ is presented in Figure 9.

	OPC	w	explanation
	2	2T f_1	jump over code for <variable>
r_1:	···	···	code generating the address of the variable on the execution stack
	20		FOR1
	2	2T f_2	jump to translation of <statement>
FLI[f_1]:	0	2A 0 A	load 0 in register A
	2	2B f_3	load address of FOR0 instruction in register B
	9		ETMP, extransmark procedure
	···	···	code for <for list>
	2	2S f_4	load address of instruction following the <for statement> into S
	27		FOR8
FLI[f_3]:	19		FOR0
	1	2T r_1 A	jump to code for <variable>
FLI[f_2]:	···	···	code for <statement>
	1	2T r_1 A	jump to code for <variable>
FLI[f_4]:			

Figure 9: The translation of ‘**for** <variable> `:=` <for list> **do** <statement>’

In Figure 9, $f_1, \ldots, f_4$ are locations in the future list filled with the appropriate (relative) object–code addresses, whereas r_1 is the (relative) object–code address given to the code for generating the address of the controlled variable on the stack (allways the second instruction of the translation of the <for statement>).

The code for loading the address of a variable to the execution stack depends, of course, on the nature of that variable:

- for a formal identifier '`v`' it is:

0	2S @`v` A	load dynamic address of `v` in register S
18		TFA, take formal address

- for a simple variable '`v`' it consists of an instruction loading the address of `v` to register B (static addressing) or S (dynamic addressing), followed by one of the instructions TRAD (OPC 14), take real address dynamic, TRAS (15), take real address static, TIAD (16), take integer address dynamic, or TIAS (17), take integer address static.
- for a subscripted variable '`v[`$i_1, \ldots, i_n$`]`' the code reads:

$\cdots$	$\cdots$	code generating the address of `v` to the execution stack
$\cdots$	$\cdots$	code generating the value of i_1 to the execution stack
$\cdots$	$\cdots$	$\cdots$
$\cdots$	$\cdots$	code generating the value of i_n to the execution stack
56		IND, indexer

The code for the <for list> is the concatenation of the codes of its constituent <for list element>s, which read:

- for an arithmetic expression E: the code generating the value of E to the execution stack, followed by FOR2 (OPC 21).
- for the 'while element' 'E **while** B':

$\cdots$	$\cdots$	code generating the value of E to the execution stack
22		FOR3
$\cdots$	$\cdots$	code generating the value of B to the execution stack
23		FOR4

- for the 'step–until element' 'E_1 **step** E_2 **until** E_3':

⋯	⋯	code generating the value of E_1 to the execution stack
24		FOR5
⋯	⋯	code generating the value of E_2 to the execution stack
25		FOR6
⋯	⋯	code generating the value of E_3 to the execution stack
26		FOR7

5.8 The compiler stack

Whereas during the prescan the compiler stack is used only to keep track of the block structure of the ALGOL 60 program, during the main scan it is used for many more purposes. In this section we give an overview of the most important applications of the compiler stack.

For pushing values on top of the stack the same compiler procedure *fill_t_list* is used as in the prescan. For popping a value from the top of the stack to a compiler variable the procedure *unload_t_list_element* is frequently used. Sometimes, however, it is done explicitely, especially if there is no interest in that value. The inspection of the top of the stack is always by explicit code.

The stack is used for the following purposes:

- to keep track of the block structure. For each block that has been entered but not yet exited the stack contains three values: a location in the future list (in which the first code address after the block has to be filled in), the length of the name list prior to block entrance, and a block–begin marker (the value 161). These are pushed to the stack, partly by means of a procedure *intro_new_block*, when encountering a declaration immediately following a delimiter '**begin**', and when dealing with the delimiters '**do**' and '**procedure**'. They are popped from the stack when a delimiter '**;**' or '**end**' indicates the end of the block.
- as discussed in Section 5.3, at context switches, to save and later restore part of the context state.
- to record the begin address of a piece of code which will be referred to at some point(s) in the sequel, in situations where the possibility of recursive constructs prohibits to save it in a global compiler variable. An example of this we met in the previous section, where the address r_1 of the code for the controlled variable of a <for statement> has to be saved for later use. Since the controlled statement can be or contain another for statement, this code address has to be saved in the stack. It is pushed when dealing with '**for**' and popped in the treatment of '**;**' and '**end**'

when it is the end of that for statement.

- to record locations in the future list in which yet unknown code addresses have to be filled in. For an example we refer to the previous section again, where location f_4 is pushed to the stack (when dealing with '**do**') and popped at the end of the for statement. Note that neither f_1 nor f_2 nor f_3 need to be saved in the stack: only when dealing with the controlled statement new for constructs can be encountered; f_1 is kept in the global compiler variable *fora*, f_2 in *forc*, and f_3 in *fora* again.
- in the transformation of expressions to polish–reversed format. This is discussed in the next section.
- to record the code words for the actual parameters of a <procedure statement> or a <function designator>. An example of the translation of a procedure statement is given in Section 5.1. The four code words are constructed during parameter analysis and pushed to the stack when dealing with the parameter separator '**,**' or the parenthesis concluding the parameter list. They are popped when dealing with that concluding parenthesis.
- to record the entry points for the code for the <designational expression>s of a <switch list> in a <switch declaration>. When the concluding semicolon is encountered a piece of object code is generated with a jump instruction to each of these entry points.

5.9 The transformation of expressions

The transformation of expressions to polish–reversed notation is based on a priority scheme. To each operator a priority is assigned according to the following table:

(	0
$:=$	2
$\equiv$	3
$\Rightarrow$	4
$\vee$	5
$\wedge$	6
$\neg$	7
$<, \leq, =, \geq, >, \neq$	8
+,(binary)−	9
(unary)−,∗,/,_:	10
$\uparrow$	11

Note that subtraction gets priority 9 whilst the unary operator for sign inversion gets priority 10.

In the transformation the code for loading operands is always generated immediately, the code for operators has possibly to be postponed until the priority of the context is sufficiently low. In case of postponement the operator is saved in the stack. In fact, pairs are pushed to the stack consisting of the operator itself and its priority (coded as $256 *$ priority + representation of the operator). An invariant of the algorithm is that the top part of the stack contains some value with priority part 0 and zero or more operators with priority part ≥ 2.

While scanning an expression from delimiter to delimiter, for each operator roughly the following actions are carried out:

- set the operator height *oh* equal to the operator's priority.
- if *nflag* = 1 (indicating that the operator was immediately preceded by an identifier or constant) then generate an instruction that loads an address into register B or S using the information found in the compiler variable *id*. The appropriate load operation for the operand is selected. If the top of stack contains one of the operators +, (binary) $-$, $*$, $/$, or _: and if its priority part is at least *oh*, that operator is removed from the stack and integrated with the selected load operation. The resulting operation code is added to the object code.
- as long as the top of the stack contains an operator/priority pair with priority part at least *oh* it is removed from the stack and the corresponding operation instruction is added to the object code.
- the current value of *dl* and its priority are pushed to the stack as a new operator/priority pair.

The first three of these actions are executed by a call of the compiler procedure *production_of_object_program* with the operator's priority as a parameter.

At the occurrence of the first delimiter *not* belonging to the expression (i.e., one of the symbols from the follow set of <expression> consisting of the symbols ')', ']', ';', '**end**', '**then**', '**else**', '**do**', '**while**', '**step**', '**until**', ',', and ':') the translation of the expression is finalized by a call of *production_of_object_program* with parameter value 1 (in some cases indirectly via a call of the compiler procedure *empty_t_list_through_thenelse*).

A delimiter '(' within an expression is pushed to the stack with priority value 0. The expression following it is thereby handled separately; at the occurrence of the corresponding closing parenthesis first that expression is finalized; thereafter the opening parenthesis is popped from the stack again.

The value at the bottom of the operator stack (having priority field 0) can be either the representation of some delimiter, like '(', '**if**', '**then**', '**else**', '**while**', '**step**', '**until**', '**begin**', '[', or the block–begin marker 161, or the switch list separation marker 160.

5.10 Designational expressions

Designational expressions occur in three roles: as element of a <switch list> in a switch declaration, as element of an <actual parameter list> in a procedure statement or a function designator, and following the delimiter '**goto**' in a goto statement. In all three roles a designational expression is translated in the same way: execution of the object code will always lead to the tranfer of control to some label. Consequently, the occurrence of a '**goto**' symbol can be ignored by the compiler but for the fact that it marks the beginning of a statement and thereby possibly the end of the declarations of a block.

The translation of an identifier '*id*' occurring in a designational expression (i.e., an identifier having $d17 = 1$ in its descriptor in the namelist, indicating that it is a label or switch identifier) depends on the delimiter immediately following it. If that differs from '[', *id* is interpreted as a label identifier and translated in one of the following ways:

- if *id* is a non–formal identifier the translation is a jump instruction. Its precise form depends on two circumstances:
  - if the label declaration precedes the first applied occurence of that label, it reads: '2T @*id* A' with OPC–value 1, where @*id* denotes the address part of the descriptor belonging to *id*. Otherwise it reads: '2T f_{id}' with OPC–value 2, where f_{id} is the location in the future list reserved for *id*.
  - if the goto statement leads to a label outside the current block the jump instruction is preceded by two instructions:
    - an instruction '2B bn_{id} A' with OPC–value 0, where bn_{id} is the blocknumber of *id*, and
    - the instruction GTA (goto adjustment, OPC 28) which caters for the necessary adaptation of execution stack and display.
- if *id* is a formal identifier the translation reads:
  - the instruction '2S @*id* A' with OPC–value 0, followed by
  - the instruction TFR (take formal result, OPC 35).

  On the basis of the code words for *id* in the block cell, TFR transfers control to the implicit subroutine for the corresponding actual parameter.

This translation of *id* is completely produced by only one call of the compiler procedure *production_of_object_program.*

If, on the other hand, *id* is followed by '[' it is interpreted as a switch identifier. The translation of '$id[E]$' reads:

- the translation of the subscript expression E in the usual way,
- the instruction SSI (store switch index, OPC 29), which stores the value of E, incremented by 1, in store location 48 (16 X 1),
- the translation of *id* as if it were a label identifier. This code will, when executed, transfer control to the very last instruction of the translation of the corresponding switch declaration which is the table jump instruction (the jump table just precedes this instruction).

The occurrence of a designational expression or a switch identifier as an actual parameter leads always to the production of an implicit subroutine. As all implicit subroutines it ends with the instruction EIS (end of implicit subroutine, OPC 13) which is never executed.

As an example consider the following ALGOL 60 program:

```
begin switch s:= AA;
  procedure p(ss); switch ss;
  goto if false then ss[1] else BB;
AA: p(s);
BB:
end
```

In Figure 10 we give the complete translation of this program.

It produces the following future list FLI:

location	contents
0	5
1	22
2	22
3	18
4	21
5	31
6	29

	OPC	w	explanation
	96		START
	2	2T 0	jump over switch declaration to code address 5
2:	2	2T 1	jump to code address 22, i.e. AA (FLI[1] = 22)
	1	2T 2 A	jump to code address 2 (for index value 1)
4:	0	1T 48	jump backwards over *store*[48] places
5:	2	2T 2	jump over procedure declaration to code address 22
6:	0	2B 1 A	load block number in register B
	89		SCC, short circuit
	3	2B 0 A	load KLI[0], i.e. 1, in register B
	34		TIRS, take integer result static
	30		CAC, copy Boolean accumulator into condition
	2	N 2T 3	jump over then–part to code address 18
	3	2B 0 A	load KLI[0], i.e. 1, in register B
	34		TIRS, take integer result static
	29		SSI, store switch index
	0	2S 161 A	load (dynamic) address of *ss* in register S
	35		TFR, take formal result
	2	2T 4	jump over else–part to code address 21
18:	0	2B 0 A	load 0 (block number of BB) in register B
	28		GTA, goto adjustment
	2	2T 5	jump to code address 31, i.e. to BB (FLI[5] = 31)
21:	12		RET, return
22:	1	2B 6 A	load code address 6, i.e. of p, in register B
	2	2T 6	jump to code address 29 (FLI[6] = 29)
24:	0	2B 0 A	load 0 (block number of s) in register B
	28		GTA, goto adjustment
	1	2T 4 A	jump to code address 4, i.e. to s
	13		EIS, end of implicit subroutine
	1	0A 24 B	parameter code word, code address 24
29:	0	2A 1 A	load 1 (number of parameters) in register A
	9		ETMP, extransmark procedure
31:	97		STOP

Figure 10: The translation of a program involving labels and switches

and the following constant list KLI:

location	contents
0	1

Furthermore the following (relative) code addresses are assigned to the label, switch and procedure identifiers:

identifier	code address
s	4
p	6
AA	22
BB	31

5.11 The central loop

The overall structure of the central loop of the main scan is rather simple: it consists of the following components:

1 a call of *read_until_next_delimiter*;
2 if *nflag* $\neq 0$ a call of either *look_for_constant* or *look_for_name*; moreover, *jflag*, *pflag*, and *fflag* are redefined as described before;
3 a case statement with a case for each of the possible values of *dl*, i.e. the delimiter found by RND.

There are, however, a few complicating factors.

- in a few cases of the case statement there is a need to inspect the next delimiter as a look–ahead symbol. Then, in the next iteration of the central loop, the call to RND has already been carried out and should be suppressed.
- at some other occasions also the second step of the main loop is suppressed. At one occasion this is obligatory: when the delimiter ']' is encountered and if after the restoration of the old context *jflag* happens to be 1 indicating that ']' ends a switch designator, *id* contains (a copy of) the desciptor of the switch identifier, which is still relevant for the generation of a piece of object code. This generation is delegated to the case for the next delimiter which is read with RNS rather than RND. At other occasions the second step of the central loop is suppressed only as a sort of short–cut because no identifier or constant should have preceded the delimiter.

- some of the cases share a piece of code. This is implemented by jumps from one case into another (and sometimes back again). A typical example of this is found in the code for delimiter '**do**', where part of the code for the delimiter ',' is executed in order to generate one of the instructions FOR2 (OPC 21), FOR4 (OPC23) or FOR7 (OPC 26), concluding the translation of the last for–list element, before continuing the code for '**do**' itself.

These factors make it hard to encode the main loop in a structured way.

Below we first present the cases for the four delimiters '$*$', '**step**', '[', and ':'.

'$*$' two subroutine calls only (cf. Section 5.9):
production_of_object_program(10);
fill_t_list_with_delimiter

'**step**' again two subroutine calls; the first one finalizes the generation of the object code for the expression preceding the delimiter '**step**' (which might be a conditional expression):
empty_t_list_through_thenelse;
fill_result_list(24{FOR5},0)

'[' we have the following components:

- if *eflag* $= 0$ then *reservation_of_arrays*;
  in a non–expression context the occurrence of '[' implies that the declaration part of the block is over. If the block contains array declarations possibly still some code for these has to be generated.
- *oflag*:= 1; *oh*:= 0;
  since a new arithmetic expression follows, initial adding operators should be interpreted as unary operators.
- save (part of) the current context to the stack: *eflag*, *iflag*, *mflag*, *fflag*, *jflag*, and *nid*.
  The stacking of *nid* is important in the case that *jflag* $= 1$, implying that the delimiter '[' is part of a switch designator.
- *eflag*:= 1; *iflag*:= 1; *mflag*:= 0;
  redefine the context such that it is that of index expressions and not that of actual parameters. Important for the interpretation of comma's.
- *fill_t_list_with_delimiter*;
  save '[' to the stack with *oh*–component 0.
- if *jflag* $= 0$ then *generate_address*;
  in case of an array identifier the delimiter '[' is part of a subscripted variable. The compiler has to generate code for loading the address of the storage function

of the array to the execution stack.

In correct programs the delimiter '[' is always preceded by an identifier.

':' Here we have two cases, one of which is selected on the basis of the context state.

1. *jflag* $= 0$: the colon is interpreted as separator in a bound pair list. The generation of the object code for the lower–bound expression is finalized and the bound pair is counted (in a global variable, no danger of recursion!):
   ic:= *ic* + 1;
   empty_t_list_through_thenelse
2. *jflag* $= 1$: the colon was preceded by an identifier with $d17 = 1$ in its descriptor, indicating that the identifier was isolated during prescan as label of a statement or as switch identifier in a switch declaration. The colon is interpreted as marking the label of a labeled statement. Since it could mark the begin of the compound tail of a block and, therefore, the end of the declarations of a block head, possibly some object code has to be generated to finalize array declarations. No further object code is needed, but, of course, the descriptor of the label identifier in the name list should be updated:
   reservation_of_arrays;
   label_declaration

The most complex case analysis is required for the delimiter ','. The following cases are distinguished:

1. *iflag* $= 1$
   The comma is interpreted as subscript separator in a subscript list.
2. $(iflag = 0) \wedge (vflag = 1)$
   The comma is interpreted as separator between for list elements in a for list.
3. $(iflag = 0) \wedge (vflag = 0) \wedge (mflag = 1)$
   The comma is interpreted as separator between actual parameters in the actual parameter list of a procedure statement or a function designator.
4. $(iflag = 0) \wedge (vflag = 0) \wedge (mflag = 0) \wedge (sflag = 1)$
   The comma is interpreted as separator between designational expressions in the switch list of a switch declaration.
5. Otherwise, the comma is interpreted as separator in the bound pair list of an array declaration.

Some cases in the case analysis in the central loop contain inner loops. These are presented in the next section.

5.12 The inner loops of the central loop

In some of the cases that are distinguished in the central loop of the main scan we find one or more inner loops. They fall apart in two classes.

In the first place there are inner loops to finalize the generation of a piece of object code after the detection of the concluding delimiter of a certain construction. Typical examples are:

- `repeat` *production_of_object_program*`(1) until not` *thenelse*;
  to enforce the completion of the code for all pending conditional constructs. We find such a loop in the cases for the delimiters '**then**', ',', ')', ']', ';', and '**end**'.
- the generation of the actual parameter code words (stored in the compiler stack), after the detection of the closing parenthesis of the actual parameter list.
- the addressing of the identifiers in an array segment of an array declaration, after detection of the closing square bracket.
- the generation of the jump table for a switch declaration from the list of begin addresses of the code for the designational expressions (stored in the compiler stack), after the detection of the concluding semicolon.

More interesting are the situations in which a piece of the source text is read, analyzed and compiled within one of the cases of the central loop. These are:

- after the detection of a string quote the string is read and transferred to the object program in portions of three characters in one word of object code.
- after the detection of the delimiter '**end**' the input string is scanned until the first occurrence of one of the delimiters ';', '**else**', or '**end**' (with one exception: if the delimiter marks the end of the program). Recall that this kind of comment is, in the prescan program, unjustly skipped by the central loop itself.
- after a type symbol ('**real**', '**integer**', or '**Boolean**') followed by an identifier the whole type list is scanned; all identifiers are added to the name list. Moreover, in the case of an inner block of the program all type declarations following the first one are analyzed at once without returning to the main loop.
- for an array declaration in the outermost block all array lists are read and analysed. According to the restrictions of X1–ALGOL, all bounds of arrays declared in the outermost block should be numbers. Their values are immediately used to construct the storage functions of these arrays which are added to the constant list.
  For an array declaration in an inner block of the program only the array identifiers of the current array segment are read and added to the name list; the bound pair list is analyzed in the central loop.

- of a procedure declaration the formal parameter part, the value part, and the specification part are completely handled after the detection of the delimiter '**procedure**'. It leads to the addition of the formal parameter identifiers to the name list and to the construction and alteration of their descriptors. Moreover, code is generated for those formal parameters that occur in the value list and that are specified as '**real**', '**integer**', or '**Boolean**'.

5.13 Store management

During the main scan of the compiler the following data structures have to be represented in store:

- the compiler stack TLI;
- the future list FLI;
- the constant list KLI;
- the name list NLI;
- the (remaining part of the) prescan list PLI.

In the latest version of the compiler two additional data structures also had to find a place in store:

- the (remaining part of the) internal representation of the source text;
- the object program RLI in compressed representation.

For the compiler stack 128 words were reserved at a fixed location as described in Section 4.2. The remaining working space of the compiler, running from store address 1933 upto 6783, was used to accommodate all other data.

The prescan list resides at the end of the available space; its length shrinkes at each block introduction, which, as explained before, transfers two sublists from the prescan list to the name list.

The compressed representation of the object code is placed at the beginning of the working space. At the start of the main scan the internal representation of the source text starts only 8 places beyond the (then still empty) object program. Luckily, during the main scan the source text is consumed whilst the object program grows. If, however, the object code is about to overwrite the source text, the latter is, together with FLI, KLI and NLI, shifted upwards over 8 places.

The future list immediately follows the source text. The constant list is initially placed 16 places beyond the (then still empty) future list and the name list 16 places beyond the (then still empty) constant list. FLI and KLI are steadily growing during the main scan, whilst NLI grows and shrinks in connection to block structure. If FLI is about to overwrite KLI both KLI and NLI are shifted upwards over 16 places; if KLI would overwrite NLI then NLI is shifted upwards over 16 places.

In this way the lists are accomodated as a kind of floating islands in a linear sea; the fact that in case of a collision the distance is enlarged by more than one place reduces the frequency of the necessary shifts and thereby the total costs of storage management. Maybe this technique was rather new in that time in which 'heaps' still had to be invented.

Before any list is shifted it is checked that by the shift the remaining part of the prescan list will not be overwritten. If that would be the case the compiler halts with error stop 6, 16, 18, or 25.

All assignments to RLI, FLI, KLI, and NLI in the compiler are executed by the invocation of a procedure in which all the necessary checks are carried out and the absolute address of the location is determined. The compiler itself only keeps track of relative positions (with respect to the begin of the lists).

5.14 Some quantitative data

In order to obtain some feeling for the performance of the compiler we collected some data of the translation of a sample program. We took the same program by Zonneveld used before.

The output of the main scan for this program can be summarized as follows:

total length of object code	2538
length of future list	192
length of constant list	84

The source program (in our lay–out) takes 185 lines (blank lines inclusive), therefore the object code has on the average 13.7 instructions per line of source text. This is a relative high number. But we should keep in mind that a simple load operation of an integer variable requires two instructions: one for loading the static or dynamic address to a register and a call to one of the routines in the complex of subroutines. This is also reflected by the fact that on the average for each delimiter found by RND 2.02 instructions of object code are generated.

From the 2538 object code instructions 1112 were generated with OPC value ≤ 3 and 1426 with OPC value ≥ 8 (i.e., a call to the complex of run–time subroutines).

The object words with OPC values ≤ 3 can be subdivided as follows:

OPC = 0	574 words
OPC = 1	88 words
OPC = 2	205 words
OPC = 3	245 words

There are 50 parameter code words, 25 words encoding strings, 189 jump instructions, 839 instructions loading a value in register A, B or S as parameter of a complex subroutine, and 9 instructions to increment the execution stack pointers for 3 procedure declarations. More specifically, we found as most frequent OPC/instruction combinations:

OPC	X1–instruction	count	frequency
0	2S ... A	454	40.8
3	2B ... A	220	19.8
2	2T ...	101	9.1
1	2T ... A	66	5.9

catering for three quarters of the cases.

The 13 most frequently generated contributions to the object program with OPC value ≥ 8 are:

OPC	name	meaning	count	frequency	accumulated
34	TIRS	Take Integer Result Static	148	10.58	10.58
33	TIRD	Take Integer Result Dynamic	129	9.05	19.42
56	IND	INDexer	129	9.05	28.47
85	ST	STore	98	6.87	35.34
14	TRAD	Take Real Address Dynamic	92	6.45	41.80
58	TAR	TAke Result	81	5.68	47.48
31	TRRD	Take Real Result Dynamic	57	4.00	51.47
9	ETMP	ExTransMark Procedure	52	3.65	55.12
18	TFA	Take Formal Address	51	3.58	58.70
16	TIAD	Take Integer Address Dynamic	35	2.45	61.15
59	ADD	ADD	35	2.45	63.60
15	TRAS	Take Real Address Static	34	2.38	65.99
19	FOR0	FOR0	32	2.24	68.23

which cater for more than two third of the subroutine calls to the complex.

Striking is the relative unimportant role of the arithmetic operations in a typical numeric program for the calculations of planetary orbits, at least at the code level. The most frequent arithmetic operation, ADD, occurs only at the 10/11th line in the list, and the total count of arithmetic operations sums up to 178, i.e. 12.48 % of the invocations of a routine in the complex.

In compacted form the object code requires only 981 words (+ 9 bits for the code word under construction), that is about 10.5 bits per instruction. We come back to this aspect in the next chapter. It overwrites gradually the input text which originally has a length of 1067 words, but in our experiments it turned out that it was never necessary to shift the yet unconsumed part of the input text upwards (together with FLI, KLI and NLI).

We also did some measurements of the number of compiler instructions executed during the main scan. This number is exclusive the instructions for encoding and storing the object string in the store in compact form (by giving *fill_result_list* temporarily an empty body) but includes the repetition of (part of) the lexical scan, especially of RND. We found in total the execution of 385 077 instructions, of which 95 058 (25 %) are spent in the lexical scan (41 611 in RNS and 53 447 in RND). This means that for the example program the main scan requires the execution of about 152 instructions per instruction of object code generated, and of about 307 instructions per delimiter analyzed.

During the main scan the name list NLI had to be shifted 5 times in order to make place for an addition to the constant list KLI, whereas KLI and NLI together had to be shifted 11 times in order to cater for the growth of the future list FLI. These 16 shifts moved altogether 2960 words (on the average 185 words per shift), which required the execution of 11840 instructions or 3% of the main scan execution time.

With a prefill of the name list of 51 words as used in our experiments the name list had a maximum length of 177 words. The maximum length of the stack was 43 words.

5.15 Some problems

The most important and inconvenient shortcoming of the X1 ALGOL 60 compiler was the almost total absence of a syntax check. Most of the checks that were carried out had to do with the proper use of the Flexowriter code (parity check, shift definitions where required). The only check that really had to do with the (context sentitive) grammar rules was the test whether all applied identifiers were declared within the context. If not the compiler stopped without even mentioning what identifier had not been declared.

Other grammatical errors lead to one of four possible forms of behaviour:

- during the prescan program the tape ran out of the tape reader, often caused by some missing '**end**' symbol. Another possible cause was the lack of some Flexowriter symbol (preferably a newline) after the last '**end**' symbol.
- the compiler just generated an incorrect object program, which passed on the problem to the execution phase. An example of this behaviour is the 'expression'

$$x + * y$$

which produces the code given in Figure 11, leaving the stackpointer AP during execution effectively unchanged.

0	2B 138 A	load static address of x in register B
32		TRRS, take real result static
0	2B 140 A	load static address of y in register B
47		MURS, multiply real static
59		ADD

Figure 11: The translation of $x + * y$

- the compiler stops with an errornumber indicating something unexplicable. An example: consider the text

begin real x; **then** x:= 1 **end**

This lead to error stop 1 in the compiler procedure *production_of_object_program* that finds an operator on the stack with a value > 151. This is caused by the fact that when dealing with the delimiter '**end**' the operator '**then**' is found on top of the stack which results in removing three words from the stack, including the '**begin**' symbol and the block–begin marker. The stack is then empty, and the next call of *production_of_object_program* inspects the word of the store below the stack. Its contents are not set by the compiler, and it depends on the history what value is retrieved. In the case of our X1–code interpreter, which initializes the whole store with the value −0, the values 255 for the operator height and 255 for the operator value are found. In the case of the Pascal version of the compiler the values 0 and 0 are found, respectively, which leads to a continuation of the compilation process beyond the last '**end**'. A lot of 'symbols' are retrieved and skipped until the code sequence '91 52 112' (for '; P **procedure**') is met. This results in error 7: unknown identifier!

- the compiler enters an endless loop. Again an example.

  begin integer i;
  procedure 0(n); **value** n; **integer** n; print(n ∗ n);
  o(5)
  end

  The loop occurs within the compiler procedure *label_declaration* (called from the code for the delimiter '**procedure**'), which tries to print the 'identifier' '0', finds that the last three bits of its encoding are zero (indicating a one–word identifier encoding) and starts to find the first non–zero part of that word, which it never will meet.

All these problems were caused by an inadequate reaction on faulty source programs, occurring, however, frequently. This does not imply, however, that all correct programs are dealt with appropriately. Apart from the problem already mentioned when dealing with the prescan program, we have also seen a case that was not compiled correctly by the main scan. The problem is demonstrated by the following program[5].

begin procedure P(a); **value** a; **integer** a;
AA: **begin integer array** A[1:100]; print(a); **goto** AA **end**;
P(10)
end

The object code produced is given in Figure 12, (with $FLI[0] = 20$ and $FLI[1] = 23$).

There are two problems here.

- First of all, there is in the code only one block for procedure P, which includes both parameter a and array A. Therefore it is impossible for the code to exit the inner block and abandon A without abandoning a at the same time. In fact, the jump to label AA does not leave any block, and in the repetition (the storage function of) array A is added to the execution stack over and over again without ever removing any of those storage functions.
- Secondly, only part of the code for declaring an array is generated: the code for generating the storage function for array A is present but the code for reserving the area for its elements is missing. The missing code (cf. Section 5.2.1) reads:

0	2S 163 A	load dynamic address of array A in register S
94		LAP, local array positioning

[5]According to the list of restrictions as reproduced in Section 1.3, 'procedure bodies starting with a label should be avoided'.

	OPC	w	explanation
0:	96		START
	2	2T 0	jump over procedure declaration (FLI[0] = 20)
2:	0	2B 1 A	load 1 into register B
	89		SCC, short circuit
	0	2S 161 A	load dynamic address of ‘a’ in register S
	16		TIAD, take integer address dynamic
	0	2S 161 A	load dynamic address of ‘a’ in register S
	35		TFR, take formal result
	85		ST, store
9:	3	2B 0 A	load static address of constant 1 in register B
	34		TIRS, take integer result static
	3	2B 1 A	load static address of constant 100 in register B
	34		TIRS, take integer result static
	0	2S 1 A	load number of arrays in register S
	91		ISF, integer arrays storage function frame
	0	2S 161 A	load dynamic address of ‘a’ in register S
	33		TIRD, take integer result dynamic
	103		print
	1	2T 9 A	jump to label `AA`
	12		RET, return
20:	1	2B 2 A	load address of procedure in register B
	2	2T 1	jump over parameter code word (FLI[1] = 23)
	3	0A 2 A	codeword for parameter ‘100’
23:	0	2A 1 A	load number of parameters in register A
	9		ETMP, extransmark procedure
	97		STOP

Figure 12: The incorrect translation of a correct program

The explanation is more subtle and is a consequence of the way in which the generation of the reservation of store for array elements is postponed until no more array declarations can follow. For that purpose there is a compiler variable *vlam*. It is set to some value $\neq 0$ for each new block encountered. It is inspected at each delimiter that implies that no (further) array declarations of the block can follow. If it is non–zero, it is set to zero and the part of the namelist corresponding to the block is scanned for the presence of value array parameters and local arrays (marked in the namelist by a descriptor with $d26 = 1$). For these the instructions to reserve the store for the array elements are generated. In the present case, *vlam* is already set to zero upon the occurrence of label AA in the text (marking that the statement part of the block is being scanned), at a moment that identifier A is not yet incorporated in the namelist. The declaration of array A is not treated as marking the start of an inner block to the procedure body, due to the presence of a block–begin marker just below the top of the stack. Consequently, *vlam* is not set to a value $\neq 0$ again and no further inspections of the namelist will take place when it is zero.

Chapter 6

The compiler output

6.1 The first version

Originally, the object program generated by the main scan was punched on 5–track paper tape. The paper tape contained[1]:

- a piece of about 50 cm of blank tape;
- an endmarker 'XCXX' (in fact, an empty cross–reference list);
- a piece of about 10 cm of blank tape;
- the 'result list', i.e. the instructions of the object program;
- the constant list, each word of the constant list given an OPC value of 0;
- a piece of about 50 cm blank tape;
- 5 numbers, i.e. the number of object words, the length of the constant list, the length of the future list, the address of the first unreserved word of the execution stack, and the begin address of the execution stack (i.e. 138), each given an OPC value of 0;
- the elements of the future list, with OPC value 1;
- the number of MCP's (library routines) called directly from the object program (with an OPC value 0);
- the places in the future list which contain the identification data of those MCP's (again with OPC value 0);
- a piece of about 50 cm of blank tape.

[1]Since the original code of the compiler seems to be lost, the information given here is largely reconstructed from the code of the loader program.

Each item, consisting of an OPC value and, in case of an OPC value ≤ 3, a 27–bit word, was punched as 2, 5, or 7 pentads in the following way:

- for an OPC value ≥ 8: 2 pentads, consisting of a parity bit, a code bit 1, and the OPC value in 8 bits;
- for an OPC value ≤ 3 and a w value corresponding to one of 10 different instruction types: 5 pentads, consisting of a parity bit, two code bits 0, a value between 1 and 10 indicating the instruction type in 5 bits, the OPC value in 2 bits, and the address part of the instruction in 15 bits;
- for an OPC value ≤ 3 and a w value not corresponding to one of the 10 instruction types mentioned above: 7 pentades, consisting of a parity bit, a code bit 0 followed by a code bit 1, three bits 0, the OPC value in 2 bits, and the w value in 27 bits.

The 10 instruction types leading to a 5 pentad encoding in the object tape are given by Figure 13.

nr	instruction type			
1		0A	0	
2		2A	0	A
3		2S	0	
4		2S	0	A
5		2B	0	
6		2B	0	A
7		2T	0	
8		2T	0	A
9	N	2T	0	
10		4A	0	

Figure 13: The 10 instruction types leading to a 5 pentad encoding

For the example program of Zonneveld we measured:

- 1426 two–pentad instructions,
- 1050 five–pentad instructions, and
- 62 seven–pentad instructions,

giving 8536 pentads for the instructions. The constant list required 502 pentads (43 five–pentads words and 41 seven–pentads words), the future list 970 pentads (187 five–pentad and 5 seven–pentads words), wereas the six numbers and the five MCP locations required 55 pentads. Together with the 4 pieces of blank tape (640 pentads 0) and the marker this gives a total of 10707 pentads requiring 428.3 seconds or about 7 minutes of punch time.

The design goal of the successor of this first output system was to reduce that punch time by at least a factor of two.

6.2 The ALD7 system

The development of the ALD7 system started in 1962. It was one of my first tasks on the institute, and my first acquaintance with a compiler. The aim was to reduce the punch time of the Dijkstra/Zonneveld compiler by at least a factor of two by means of two measures:

- punching heptads in stead of pentads (the tape punch used seven–track paper tape); this alone could reduce the length of the code on paper tape by roughly a factor 1.4;
- using a shorter encoding of the information, applying short code for frequently occurring pieces of information; another length reduction of a factor 1.4 would suffice.

The hope was that the nescessary modifications would concentrate at the periphery of the compiler only. The most extreme possibility in this respect was to encode the pentads as produced by the original version: that required the adaptation of the routine that offered the pentads to the tape punch. A measurement on the frequency distribution of pentads in some object tapes showed that a Huffmann encoding thereof would not lead to the required length reduction.

Therefore we had to go one level deeper into the compiler, to the compiler routine *fill_result_list*. Frequency measurements (again on some object tapes) of the occurence of instructions, both for OPC ≤ 3 and OPC ≥ 8, showed that it was possible to attain the required shortening by relatively simple means, allowing a fast encoding and decoding algorithm. We used one bit to discriminate between instructions with OPC ≤ 3 and those with OPC ≥ 8. The latter were encoded in 3, 4, 5, 6 or 9 additional bits, depending on their frequency of occurrence.

For the instructions with OPC ≤ 3 the 15 bits of the address parts were split into three portions of 5 bits, each of which was encoded according to its own frequency distribution. The 12 bit function part was encoded together with the OPC value itself: for the 19 most frequently occurring combinations a 2–, 3–, or 6–bit additional value was used, the other combinations were encoded in the same way as an address part together with a special 6–bit escape code.

The full details of the encoding can be found in Appendix D.

During the design period it was suggested (came the suggestion from L.A.M. Meertens?) that if the tape was punched in such a way that it could (and should) be read and decoded in the backwards direction, the amount of tape handling in the program loading phase could be reduced greatly. This requires some further explanation.

The object tape consists essentially of two sections:

- the result list and the constant list, and
- some numbers and the future list.

The problem was that they have to be produced in this order – due to the fact that those numbers and the contents of the future list are known only at the end of the compilation process –, but have to be loaded in the opposite order – a.o. since substitutions of references to the future list (OPC value 2) by the value found there are carried out immediately during reading of the result list –. Moreover, the reading of the so–called Cross–Reference List CRF, containing information about the mutual use of MCP's library routines (see Chapter 7), had to be inserted in between. By inserting the contents of the CRF in the loader (with the disadvantage that when the contents of the library were updated also a new loader tape had to be produced) and reading the object tape in the backwards direction the latter could be read at one stroke.

In the ALD7 version the object tape consisted of:

- a piece of 50 cm blank tape,
- a punching 124, followed by a punching 30 as end combination,
- a piece of blank tape of 6.25 cm,
- a punching 127 as section end,
- the following bitstring, cut into pieces of 27 bits, to each of which a parity bit (for odd parity) is added and which are punched in 4 heptads:
  - the result list,
  - the constant list, each word of it given an OPC value of 0,
  - the places in the future list which contain the identification of the MCP's called directly from the object program, each encoded as an address,
  - the number of those MCP's, encoded as address,
  - the future list, each word of it given an OPC value 1,
  - 5 numbers, i.e. the begin address of the execution stack (i.e. 138), the address of the first unreserved word of the execution stack, the length of the future list, the length of the constant list, and the number of object words, each encoded as an address,
  - a bit 1, as marker of the begin of the information,
  - enough bits 0 to complete the current group of 27 bits,

- a punching 30, indicating the begin of a section,
- a piece of 50 cm blank tape.

During loading the end combination enforced a machine stop, giving the opportunity to insert the library tape into the tape reader.

The changes to the original compiler were relatively small. Routine *fill_result_list* had to be rewritten completely and two subroutines for subtasks were added: *address_coder* and *bit_string_maker*. The latter had functionally two arguments: n, the number of bits to be added to the bit string, and w, the bits themselves, but for practical purposes these two argument values were packed into one parameter: $1024*n+w$. Quite often this parameter value was taken from a table. All additions to the bitstring used *bit_string_maker* and it was inside that routine that a parity bit was added to each 27 bits of the bitstring and that the result was punched in portions of 7 bits. Furthermore the compiler code following the main scan had to be adapted to the new order and lay–out of the output tape.

Of course also a new loader had to be written. Moreover programs were written to recode the library tape in the same format as the object tape and to make a table version of the library cross–reference tape.

Although developed for shortening the punch time of the compiler, that aim was soon superseded by the arrival of a fast tape punch (Creed 3000). The shorter length of the object tape and the increased ease of tape handling in the loading phase, however, retained their value. Also the library tape in the ALD7 version used the Huffmann encoding and heptads (as opposed to pentads in the original system) and was considerably shorter than before.

For the example program of Zonneveld we measured a bitstring of in total 33318 bits, punched in 4936 heptads. Together with the additional punchings this leads to 5365 heptads, punched in 214.6 seconds or about 3.5 minutes of punch time (on the old tape punch).

6.3 The load–and–go version

In the fall of 1963 the ALD7 could already be replaced by a load–and–go version of the compiler.

The original ALGOL system for the X1 was designed to operate in a 4K word memory machine. The compiler was about 2K words long, and only 2K words remained to be used as working space, for the compiler stack, prescan list, the future list, the constant list, the name list, and the prescan list. The compiler code was positioned at the high end of the store. For program execution it was overwritten by the complex of run–time subroutines (again about 2K long). The loader was positioned at the low end of the store. During program loading the object code (the constant list included) was positioned adjacent to the complex and the library routines (used by the program) in front of that. During execution the loader was overwritten by the execution stack.

In the mean time the store size was extended to 12 K. The additional space was used during program execution, but until then hardly for program compilation. The compiler was positioned from 6K to 8K, such that the run–time routines (moved to the area from 10K to 12K) could reside in store during compilation, and substantially longer programs could be compiled. That situation was retained in the first version of the ALD7 compiler.

The first real application at compile time of the increased store size was the storage of the source text during the prescan phase of the compiler, thereby eliminating the need to read the source text twice. It was implemented in the second version of the ALD7 system. After that the idea was born to store also the object code as produced by the compiler (in its compacted form!) in the memory instead of punching it, and to integrate the loader as a third phase of the compiling proces.

For its implementation only a suitable memory management had to be devised. The 'system tape' now contained the compiler, the loader, the complex, the cross–reference list, and, in a second release, part of the library routines[2]. The following store lay–out was used:

- after system loading (addresses in the number system with base 32, therefore 01–00–00 is just 1K):
  - 00–07–00 / 00–18–26: loader program
  - 00–19–15 / 00–22–02: cross–reference list
  - 00–29–00 / 01–13–18: library selection
  - 06–25–00 / 06–29–10: prefill of the name list
  - 07–04–02 / 09–28–00: compiler program
  - 09–29–21 / 11–31–00: complex ALD
- during prescan and main scan the area from 01–13–18 to 06–20–00 is used for object string, source text, future list, constant list, name list and prescan list, in this order as described earlier. The compiler stack is located from 00–25–00 to 00–29–00.
- in the transposition from main scan to loader the constant list is moved to its final

[2]This system tape consisted of a good 6000 words, punched in 4 heptades a word. Its length was therefore slightly more than 60 m, its reading time (by means of a special fast reading program for binary tapes) about 25 s.

place, adjacent to and in front of the complex[3]. Moreover by consultation of the namelist, the future list, and the cross–reference list two lists of 128 places each are constructed indicating the directly and indirectly used library routines and their loading addresses. After that the only relevant parts of the contents of the store are the loader program, the two library lists, the library selection, the objectstring, the future list, the constant list, the complex, and some numbers saved in the working space of the loader.

- during program loading:
  - 00–07–00 / 00–18–26: loader program
  - 00–19–15 / 00–23–15: list of library use
  - 00–25–00 / 00–29–00: list of library use
  - 00–29–00 / 09–29–21:
    - remainder of library selection and object string at the low end,
    - loaded part of library selection, object program, and constant list at the high end,
    - future list somewhere in between.

  The loading proceeds in backwards order. Whenever the loaded program reaches the future list's end, the latter is moved downwards against the remainder of the object string.

In the coding much profit was taken from the ALD7 compiler. In the main scan part only routine *make_bit_string* had to be rewritten in order to store each portion of 27 bits instead of punching them. Of course also some initializations and the code following the main scan had to be rewritten. The existing loaders could be used as blue–print.

The load–and–go compiler was put into operation in november 1963.

[3] by a piece of compiler code located from 09–13–20 to 09–13–28, apparantly never overwritten by it.

Chapter 7

The library system

In the foregoing chapters we refered to the library system already a number of times. Here we give some more detailed information.

In the ALGOL 60 system for the X1 a number of procedures and functions were incorporated. Part of them were the standard functions mentioned in the Revised Report: *abs, sign, sqrt, sin, cos, arctan, ln,* and *exp*. Other ones were added for input/output (for the console typewriter, the tape punch, and, at a later stage, a plotter). Moreover, some frequently used algorithms were gradually added to the library, for finding zeros, solving linear equations, computing special functions, etc. All of them could be used in ALGOL programs without declaration or any other way of signalling their usage. All their names were entered in a list (added to the compiler code) which was copied to the name list *NLI* at the start of the main scan.

These procedures and functions were implemented in two different ways:

- *abs, sign, sqrt, sin, cos, ln, exp, entier, read, print, TAB, NLCR, XEEN,* and *SPACE* are included in the complex of run–time subroutines. They have each an OPC number and are treated as operators changing the top of the execution stack. Consequently, they cannot be used as an actual parameter in a procedure statement or function designator, nor can the function identifiers among them be used in a procedure statement (since there is no mechanism to remove the function result from the stack). The first *nlscop* words from the prefill of the name list belong to these procedures and functions. As all routines in the run–time complex these operators are coded in X1 code.
- All other procedures and functions are included in the library proper and called MCPs (for Machine Code Procedures). They are written in some extension of X1 code to

be discussed below, and programmed in such a way that there were no restrictions in usage whatsoever: they could be used as if declared in the outermost block of the ALGOL program. Originally they were assembled and punched in object code format on paper tape, the library tape. At the end of program loading this tape was read and the routines that were directly or indirectly used by the program were selectively added to the object program. The routines could refer to one another (even recursively), and therefore the need of a program was the transitive closure (with respect to the use relation) of the routines that were called directly from the source program. Some MCPs were 'anonymous'. Not having an identifier that could be referred to in an ALGOL 60 program, they could be used only indirectly by other MCPs. The names of the non–anonymous MCPs were collected in the second part of the name–list prefill.

By means of an example we like to give an impression of the nature of the code of an MCP. We present the text of 'AP 109': the MCP *RUNOUT*. Its task is to punch 81 blank heptades on the output tape. Its code is reproduced in Figure 14.

```
DPZE          16    X 0             MCP number of RUNOUT is 16
DPZF          20    X 0             MCP number of PAS1 is 20


DN           + 8                    RUNOUT has 8 instructions
DI     0A      0   ZE 0             MCP number of RUNOUT

X0X    2B      1          A         blocknumber = 1
X89                                 SCC, short circuit
X0X    2A     80          A         number of zeros
X0X    6A      0    X 0             set counter
X0X     2S   128          A         blank, with punch mark
X3X    6T      0   ZF 0   2         call PAS1, i.e., punch!
X1X    4T      4    X 0   0 E       decrement counter and jump if ≥ 0
X12                                 RET, return
X
```

Figure 14: code of the MCP *RUNOUT*, AP 109

The first two lines of Figure 14 define the MCP numbers of MCP *RUNOUT* and of MCP *PAS1*. The latter is an anonymous MCP (whose name is not in the prefill of the name list). Then follows the part that constitutes the MCP itself: first two numbers, the MCP length and its number (the latter encoded as an instruction of which the function part happens to consist of 12 bits zero) and the (in this case) 8 instructions of the MCP. These are either an X1 instruction preceded by an OPC code 0, 1, or 3, or a call to one of the run–time routines of the complex, indicated by its OPC number (≥ 8). The last line contains an end marker for the last instruction.

So we see that the body of an MCP is in fact a mixture of X1 code proper and 'connectors' to its ALGOL environment. Its starts by the standard two instructions for all procedures, whether declared in the ALGOL program or member of the library, and ends with the standaard return instruction for all procedures. The X1 instructions themselves have an OPC code, indicating whether at load time the address part of the instruction should be kept unchanged (OPC 0), whether the begin address of the MCP itself should be added to it (OPC 1), or (OPC 3) whether it should be replaced by the begin address of some other MCP (the number of which is given by the given address part). OPC value 2 never occurs in MCPs. Anonymous MCPs do not need connector code to the ALGOL environment.

The library tape contained all MCPs in object code format (as described in Section 6.1) and was concluded by an end marker (the pseudo MCP length 16383). To it corresponded a separate cross–reference tape CRF with the following contents:

- for each MCP:
    - a punching 31,
    - the MCP length in 3 pentads (with odd parity),
    - the MCP number, in 2 pentads (with odd parity),
    - a list of the numbers of those other MCPS that call this MCP directly or indirectly (each in 2 pentads with parity),
    - the pseudo MCP number 511 as end marker for the list;
- a punching 31,
- the pseudo MCP length 16383 as end marker for the CRF tape.

The cross–reference table was used during program loading. Details of its use are described in Section 8.1.

The program to translate the assembly code of MCPs to object code format had as input the assembly code of one or more MCPs and the most recent version of the cross–reference tape. It produced the extension to the library tape and an updated version of the cross–reference tape. MCPs that called one another recursively should be translated together.

For the transition to the ALD7 system (discussed in Section 6.2) two programs were written to recode both the library tape (to the new object tape format) and the CRF tape. Since the latter was to be incorporated in the ALD7 program loader, it was punched in standard X1 binary tape format.

In the load–and–go versions of the compiler it was possible to incorporate some of the most frequently used MCPs directly in the compiler tape, simplifying tape handling for programs with low MCP demands. We come back to this point in the next chapter.

Chapter 8

Program loading

The main task of program loading is, of course, loading into store the compiled ALGOL 60 source program and all the library routines it uses directly or indirectly, thus delivering a program ready for execution. In order to fulfill that task, it has to do some other tasks.

First it has to determine which library routines are needed. It does so from a list of library routines that are called directly from the source program and augments this to a list of all library routines needed with the help of information from the cross–reference list.

Secondly, it has to determine where object program and library routines will be placed, by computing the begin addresses of both the object code and of all the library routines used. It does so using the length of the result list RLI, the length of the constant list KLI, and the lengths of the library routines as given by the cross–reference list.

Thirdly, while loading the object code and the code of the library routines, it has to deal with the OPC code of each instruction. If that OPC code is at least 8, it defines the instruction by itself: the instruction should be taken from the OPC table. If that OPC code is 2 (occurring in the object program only), it should replace the address part by an address taken from the future list FLI. In case of an OPC code 3 either the begin address of the constant list should be added (for an instruction in the object program) or the address part should be replaced by the begin address of an MCP. An OPC value of 1 leads to the addition of the begin address of either the object program (for the object code) or the current MCP to the address part of the instruction.

Fourthly, some minor adaptations are aplied to the object program. In case of an OPC value of 2, if bit d_{17} of the instruction is 1 it is set 0, otherwise bit d_{19} is set 1. Probably these indicate some 'maintenance' actions to the original compiler that easiest could be done in the loader (rather than in the compiler). A number of jumps in the compiler, certainly all jumps that refer to a location in the future list, are coded by the compiler as indirect jumps. Maybe it was originally planned to have the future list in store during execution and to lead those jumps via it. The setting of d_{19} changes the jumps into direct jumps, and the substitution of the location in the future list by its contents then makes the presence of the future list during program execution superfluous. The resetting of d_{17} has, according to a comment in the (revised) loader, to do something with a recoding of actual parameter code words (PORDS), but its meaning is not clear.

Finally, at the end of the loading phase, the store is prepared for a reproducible program execution by filling the whole working space by -0 (this had also the advantage of stopping the machine if, by loosing proper control, it tries to execute an unused word of working space as an instruction).

8.1 The original loader program

In the version that was documented together with the complex of subroutines (since it is referring to 'the older version' it probably is the second release of the loader) the object code was loaded in front of the complex (that started at location 10299 = 10 – 01 – 27). First the lengths RLSCE of the result list RLI and KLSCE of the constant list KLI were read from tape, and from it the begin address of the object code RLIB = 10299 – RLSCE – KLSCE (truncated downwards to a multiple of 32) and KLIB = RLIB + RLSCE were computed. Thereafter length FLSCE of the future list and address GVC0 were read. The begin address FLIB of the future list was taken as 608 (= 00 – 19 – 00), and the future list was read and loaded from that point. Note that due to the OPC coding 1 of the future list words each of these was increased by RLIB.

Next the use–list MLI, running from location 480 (= 00 – 15 – 00) to 607 (00 – 18 – 31) was initialized with 128 zeros. Then the number RNB of directly called MCPs was read, followed by the RNB future list locations were the MCP numbers could be found. For these MCPs the corresponding positions of MLI were filled with – (FLI location + FLIB), indicating their (direct) use. The begin address MCPE of the last located entity was initialized to RLIB.

The X1 stopped in order to load the CRF tape. This tape was read until its end marker (the pseudo MCP length 16383). For each MCP in the library its length was read. Thereafter the list of 'users' (starting with the MCP itself) was read and if at least one of them was wanted (the corresponding MLI positions different from 0), the MCP itself was wanted. In that case MCPE was decreased by the MCP length, and its new value was copied to MLI as begin address of the MCP. Moreover, in the case of direct use from the object code (as seen from a negative old value in MLI) that begin address was also filled in in the corresponding location in the future list.

After the processing of the cross–reference tape all the nescessary addresses (of RLI, KLI, and all MCPs that are needed) were known, and the actual loading could start. De X1 stopped for loading the (first part of the) object tape. After reading and loading RLI and KLI the begin address of the object program RLIB was typed on the console typewriter in the number system with base 32 (xx xx 00) Also MCPE was typed in the same way. The working space (from 680 to MCPE) was filled by −0 and the X1 stopped for loading the library tape. From this tape all MCPs were read, but only those that were used (as indicated in MLI) were loaded from the begin address as given by MLI. At the end of the library tape the program was ready for execution and the X1 stopped anew, now giving opportunity to load a potential data tape.

In the implementation the main subroutine is *LIL* (Read List) for reading and placing a list of instructions. *LIL* uses subroutine *RBW* (Read Binary Word), which builds the next instruction from 2, 5, or 7 pentades, incorporating its OPC–value. *RBW*, in turn, uses subroutine *RNP* (Read Next Pentade).

In this (second) version of the loader no use is made of the interrupt system for the tape reader. This suggests that it was written after the arrival of the fast tape reader. It runned reasonably fast. One inefficiency still is that during the processing of the library tape the contents of each MCPs is decoded to instructions independent of whether that MCP is actually needed or not. We remediated that in the load–and–go system.

8.2 The loader for the ALD7 system

Apart from a different decoding of the object tape and the library tape and the fact that the cross–reference information is taken from store rather than from tape, the differences are not very big.

Again the main subroutine is *LIL* for reading and placing a list of instructions, in their turn read by subroutine *RBW*. It is in *RBW* that the bitstring is decoded, thereby using additional subroutines *ML* (Read Mask), for decoding the function part of an instruction (with OPC ≤ 3), *ADD* (Address Decoder), for decoding the address part of an instruction), and *RBS* (Read Bits), for reading a front portion of the bit string (the length of which specified in its parameter). It is only in *RBS* that heptades are read from paper tape and that the parity of each group of 4 heptades is checked to be odd.

RBS operated roughly in the following way:
It maintained, in one word of its working space, a number of 'bits in stock'. Those bits, at least 21 (and, of course, at most 27), were positioned at the most significant part of the word, the first bit at position d_{26} (that bit was therefore easily inspected by testing the sign of the stock word). Moreover, the number of bits in stock was registered, and as soon as, by a call of *RBS*, the stock becomes shorter than 21 bits, a heptade is read from tape and added to the low end of the stock word. The logical sum of each group of 4 heptades is formed, and checked for parity when the first heptade of the next group is read. In that case only (the most insignificant) six bits of the new heptade are added to the stock, otherwise all seven bits are added. *RBS* is initialized by setting the number of bits in stock equal to zero, by skipping blank tape and the first non–blank heptade (requiring that it has the value 30), by loading 4 heptades, and by calling RBS (each time for 1 bit) until a bit 1 is obtained.

In the main part first RLSCE and KLSCE are read (by calls of *ADD*), RLIB and MCPE are computed, and FLSCE and GVC are read (again by *ADD*). Next the future list is read by *LIL*. Then RNB, the number of directly used MCP's is read (by *ADD*), and, if different from 0, the use–list MLI is initialized, the RNB references to the future list containing their specifiction are read (by *ADD* again) and incorporated in MLI, and the cross–reference list CRF is read from store and processed. For reading CRF a subroutine *LC* (Read Cross Reference), yielding an MCP length or number, is used.

Now the result list RLI and the constant list KLI are read (by a call of *LIL*). RLIB is typed on the console typewriter.

Next the MCPs are loaded in the following way:
Each time an end marker (i.e., pseudo MCP length 7680) is found, it is checked whether all MCPs have been read. If so, MCPE is typed, the working store for execution is cleared, and the X1 stops with stop nr 3–7, ready for program execution. If not, the X1 stops with stop nr 3–6, indicating that a (next) MCP tape should be entered in the tape reader. This organization makes it possible both that if no MCPs are used at all the reading of the MCP tape(s) can be skipped, and that, if the user removes the end marker from his object tapes and glues a copy of an MCP tape to it, the loading can proceed without intermediate stop. If an MCP length less than 7680 is found, the MCP number is read, and if it is used, that MCP is loaded by a call of *LIL*. If, however, MLI indicates that it is not used at all, the MCP is skipped (although still the instructions are decoded by calls of *RBW*).

Allthough the cross–reference list is now build–in in the loader, the user had the possibility to load his own version by means of the standard input program of the X1 using directive DW followed by binary encoded tape (as if directive DB had been read). This did, however, not alter the contents of the name list prefill, and, to the best of my knowledge, this facility never was used.

8.3 The loading phase of the load–and–go compiler

In a previous section (Section 6.3) much information has already been given about the store management of the load–and–go version. We discuss here the main differences from the ALD7 loader.

The structure of the loading phase is that of the ALD7 loader. The main difference is in the subroutine *RBS* (Read Bits), which now is capable to obtain its bits from two sources: from store, for the object program and for part of the library, and from tape, for the part of the library not in store. It is initialized in three different ways:

- the *RBS* switch is set to 'reading from store', the 'bits in stock' word is partly taken from the bits in stock of the *make_bit_string* routine of the main scan, and completed from store;
- the number of bits in stock is set to zero, the bit stock is completed from store (thus requiring a full word), and *RBS* is called for the next bit until it delivers a bit 1;
- the *RBS* switch is set to 'reading from tape', the number of bits in stock is set to zero, blank tape and a heptade 30 are skipped, the bit stock is completed from tape, and *RBS* is called for the next bit until it delivers a bit 1. In fact this is almost the initialization of *RBS* from the ALD7 loader.

Again *RBS* keeps a stock of at least 21 bits. It is supplemented by 6 (1 out of 4 times) or 7 (3 out of 4 times) bits in order to keep this invariant. In case of reading from store they are taken from $d_{26} - d_{21}$, $d_{20} - d_{14}$, $d_{13} - d_7$, and $d_6 - d_0$, successively, of a word from store.

Another difference to the ALD7 loader is in the table of MCP use. In stead of one table there are now two of such tables. At the end of the main scan, before switching to the loader program, the table MLI is cleared, and from the initial namelist part (from location *nlscop* to $nlsc0 - 1$) the locations in the future list are isolated for used MCP's (having a descriptor with bit $d_{15} = 0$). From those locations in the future list the MCP numbers are isolated and at the corresponding places of MLI the values – (FLIB + relative FLI–address) are filled in. Then the cross–reference table from store is used to determine the secondary needs and to compute and store the begin addresses of all used MCP's in MLI and, for primary use, also in the future list.

After loading the result list RLI and the constant list KLI (and the typing of RLIB) a copy is made of MLI (it overwrites the area reserved for the cross–reference table, thereby deleting that cross–reference table). During the loading of MCPs the copy is consulted to see whether an MCP is needed, and if so, to find the appropriate place. After loading of such a needed MCP, the number of needed MCPs is decremented by one and the entry in the copy of MLI is cleared (indicating that that MCP is no longer needed). It is, however, maintained unaltered in MLI itself, and it is that list that is used in processing an OPC value of 3.

By this organization it is possible to have several copies of the same MCP in memory and/or on paper tape, only one of which (the first one encountered) is loaded when needed. It also gave users the possibility to load their own version of an MCP (provided it had the length as given by the cross–reference table) by reading a private MCP tape prior to the standard one. It is again unknown to me whether this facility was ever used.

In order to accellerate the loading of MCPs, unused MCPs were no longer decoded by *RBW* (Read Binary Word), but skipped without any processing. In case of reading from store successive words from store were skipped until a fixed end pattern was found (d_{26} through d_{21} one, d_{20} through d_0 zero), in case of reading from paper tape by skipping heptades until two successive blanks were encountered. Prior to the processing or skipping of an MCP, *RBS* was reinitialized in the second or third way as specified above.

Again we give some figures for the sample ALGOL program of Zonneveld dealt with already many times. It uses 5 MCP's, all directly invocated by the program: MCP '*SUM*', '*PRINTTEXT*', '*FLOT*', '*FIXT*', and '*ABSFIXT*'. The figures are measured using a version of the load–and–go compiler in which the part of the library assembled from store contained 8 MCP's, occupying 408 words of store.

length of result list RLI	2538
length of constant list KLI	84
number of MCP instructions loaded	305
instructions executed during prescan	292810
instructions executed during main scan	531378
instructions executed during program loading	268641
instructions executed for store clearing	14161
total number of instructions	1106990
estimated execution time for prescan	15.5 s
estimated execution time for main scan	26.6 s
estimated execution time for program loading	13.4 s
estimated time for store clearing	0.7 s
total estimated execution time	56.3 s

In these times the typing time for the console typewriter is not taken into account; it could have slowed down the compiler, but in view of the limited output to that typewriter for the current program the effect is neglectible. Note that for this program the operator was not able to rewind both the source tape and the system tape during compilation.

Chapter 9

The Pascal version of the compiler

The Pascal program presented in this chapter is a back–engineering of the X1 code of the load–and–go version of the Dijkstra/Zonneveld ALGOL 60 compiler for the X1. It has been structured in the following way.

There are three main procedures, each representing a phase of the compiling process: '*prescan*', '*main_scan*', and '*program_loader*'. All procedures that are called exclusively from one of these main procedures are declared locally to the one that uses it. A procedure that is shared by two or more of these main procedures is declared globally preceding the main procedure that textually contains its first applied occurrence. We arrived at the following program lay–out:

lines 60 – 324:	the lexical scan routines. Procedure *read_until_next_delimiter* is called from both procedure *prescan* and from procedure *main_scan*.
lines 325 – 327:	procedure *fill_t_list*, storing its parameter on top of the compiler stack.
lines 328 – 436:	procedure *prescan*.
lines 437 – 570:	some procedures shared by procedure *main_scan* and the main program, in which, before calling procedure *main_scan*, the block administration for the outermost block is created and the instruction 'START' is generated.
lines 571 – 1516:	procedure *main_scan*.
lines 1517 – 1818:	procedure *program_loader*.
lines 1819 – 1992:	the main program.

The program contains some output statements not occurring in the X1 code. Some of these, placed between braces, are now comment but were previously used to inspect intermediate results.

The table given in Figure 15 can be used to find the declaration of a procedure.

procedure name	line	procedure name	line
address_coder	484	new_block_by_declaration1	674
address_decoding	1601	next_ALGOL_symbol	82
address_to_register	705	offer_character_to_typewriter	618
augment_prescan_list	351	prepare_read_bit_string1	1581
bit_string_maker	462	prepare_read_bit_string2	1587
block_introduction	355	prepare_read_bit_string3	1592
complete_bitstock	1533	prescan	328
empty_t_list_through_thenelse	866	procedure_statement	767
do_in_t_list	871	production_of_object_program	781
fill_constant_list	590	production_transmark	778
fill_future_list	580	program_loader	1517
fill_name_list	692	read_binary_word	1661
fill_output	606	read_bit_string	1571
fill_prescan_list	331	read_crf_item	1723
fill_result_list	505	read_flexowriter_symbol	60
fill_t_list	325	read_list	1707
fill_t_list_with_delimiter	577	read_mask	1630
generate_address	715	read_next_symbol	178
intro_new_block	459	read_until_next_delimiter	211
intro_new_block1	455	reservation_of_arrays	726
intro_new_block2	437	reservation_of_local_variables	698
label_declaration	622	stop	54
logical_sum	1522	test_bit_stock	1700
look_for_constant	899	test_first_occurrence	664
look_for_name	881	thenelse	856
main_scan	571	typ_address	1703
new_block_by_declaration	679	unload_t_list_element	603

Figure 15: location of the procedure declarations of the Pascal version

In the X1–code version of the compiler as given in the next chapter each component (subroutine, table, set of global variables, constant list) has its own address, characterized by two 'paragraph letters'. For example, the subroutine '*read_flexowriter_symbol*', given in the Pascal version by lines 60 through 81, has in the X1–code version addresses 0 LK 0 through 31 LK 4. In order to link the two versions of the compiler together we give for each component in the Pascal text systematically the two paragraph letters of the corresponding part in the X1–code version by means of a comment. See e.g. line 60 of the Pascal version, mentioning paragraph LK in the comment '`{LK}`'.

```
program X1_ALGOL_60_compiler(input,output,lib_tape);

const d2  =         4;
      d3  =         8;
      d4  =        16;
      d5  =        32;
      d6  =        64;
      d7  =       128;
      d8  =       256;
      d10 =      1024;
      d12 =      4096;
      d13 =      8192;
      d15 =     32768;
      d16 =     65536;
      d17 =    131072;
      d18 =    262144;
      d19 =    524288;
      d20 =   1048576;
      d21 =   2097152;
      d22 =   4194304;
      d23 =   8388608;
      d24 =  16777216;
      d25 =  33554432;
      d26 =  67108864;
      mz  = 134217727;

      gvc0 =      138;   {0-04-10}
      tlib =      800;   {0-25-00}
      plie =     6783;   {6-19-31}
      bim  =      930;   {0-29-02}
      nlscop =     31;
      nlsc0 =      48;
      mlib =      800;   {0-25-00}
      klie =    10165;   {9-29-21}
      crfb =      623;   {0-19-15}
      mcpb =      928;   {0-29-00}

var tlsc,plib,flib,klib,nlib,
    rht,vht,qc,scan,rfsb,rnsa,rnsb,rnsc,rnsd,
    dl,inw,fnw,dflag,bflag,oflag,
    nflag,kflag,
    iflag,mflag,vflag,aflag,sflag,eflag,jflag,pflag,fflag,
    bn,vlam,pnlv,gvc,lvc,oh,id,nid,ibd,
    inba,fora,forc,psta,pstb,spe,
    arra,arrb,arrc,arrd,ic,aic,rlaa,rlab,qa,qb,
```

```
    rlsc,flsc,klsc,nlsc: integer;
    bitcount,bitstock: integer;
    store: array[0..12287] of integer;
    rns_state: (ps,ms,virginal);
    rfs_case,nas_stock,pos: integer;
    word_del_table: array[10..38] of integer;
    flex_table: array[0..127] of integer;
    opc_table: array[0..112] of integer;

    rlib,mcpe: integer;

    lib_tape: text;

    ii: integer;

procedure stop(n: integer);
{emulation of a machine instruction}
begin writeln(output);
  writeln(output,'*** stop ',n div d5:1,'-',n mod d5:2,' ***');
  halt
end {stop};

function read_flexowriter_symbol: integer;                          {LK}
label 1,2;
var s,fts: integer;
begin
  1: read(input,s);
     if rfsb = 0
     then if (s = 62 {tab}) or (s = 16 {space}) or (s = 26 {crlf})
          then goto 2
          else if (s = 122 {lc}) or (s = 124 {uc}) or (s = 0 {blank})
               then begin rfsb:= s {new flexowriter shift}; goto 1 end
               else if s = 127 {erase} then goto 1
               else stop(19) {flexowriter shift undefined};
  2: fts:= flex_table[s];
     if fts > 0
     then if rfsb = 124
          then {uppercase} read_flexowriter_symbol:= fts div d8
          else {lowercase} read_flexowriter_symbol:= fts mod d8
     else if fts = -0 then stop(20) {wrong parity}
     else if fts = -1 then stop(21) {undefined punching}
     else if s = 127 {erase} then goto 1
     else begin rfsb:= s {new flexowriter shift}; goto 1 end
end {read_flexowriter_symbol};
```

```
function next_ALGOL_symbol: integer;                                {HT}
label 1;
var sym,wdt1,wdt2: integer;
begin sym:= - nas_stock;
  if sym >= 0 {symbol in stock}
  then nas_stock:= sym + 1{stock empty now}
  else sym:= read_flexowriter_symbol;
1: if sym > 101 {analysis required}
  then begin if sym = 123 {space symbol} then sym:= 93;
         if sym <= 119 {space symbol, tab, or nlcr}
         then if qc = 0
              then begin sym:= read_flexowriter_symbol;
                     goto 1
                   end
              else
         else if sym = 124 {:}
              then begin sym:= read_flexowriter_symbol;
                     if sym = 72
                     then sym:= 92 {:=}
                     else begin nas_stock:= -sym; sym:= 90 {:} end
                   end
         else if sym = 162 {|}
              then begin repeat sym:= read_flexowriter_symbol
                     until sym <> 162;
                     if sym = 77 {^} then sym:= 69 {|^}
                     else if sym = 72 {=} then sym:= 75 {|=}
                     else if sym = 74 {<} then sym:= 102 {|<}
                     else if sym = 70 {>} then sym:= 103 {|>}
                     else stop(11)
                   end
         else if sym = 163 {_}
           then begin repeat sym:= read_flexowriter_symbol
                  until sym <> 163;
                  if (sym > 9) and (sym <= 38) {a..B}
                  then begin {word delimiter}
                         wdt1:= word_del_table[sym] mod 128;
                         if wdt1 >= 63
                         then sym:= wdt1
                         else if wdt1 = 0
                         then stop(13)
                         else if wdt1 = 1 {sym = c}
                         then if qc = 0 {outside string}
                           then begin {skip comment}
                                  repeat sym:= read_flexowriter_symbol
                                  until sym = 91 {;};
```

```
        sym:= read_flexowriter_symbol;
        goto 1
       end
    else sym:= 97 {comment}
   else begin sym:= read_flexowriter_symbol;
       if sym = 163 {_}
       then begin repeat sym:=
                read_flexowriter_symbol
          until sym <> 163;
          if (sym > 9) and (sym <= 32)
          then if sym = 29 {t}
           then begin sym:=
                  read_flexowriter_symbol;
                 if sym = 163 {_}
                 then begin repeat
                      sym:=
                      read_flexowriter_symbol
                     until sym <> 163;
                     if sym = 14 {e}
                     then sym:=  94 {step}
                     else sym:= 113 {string}
                    end
                 else stop(12)
                end
           else begin wdt2:=
                  word_del_table[sym] div 128;
                 if wdt2 = 0
                 then sym:= wdt1 + 64
                 else sym:= wdt2
                end
          else stop(13)
         end
       else stop(12)
      end;
   repeat nas_stock:=  - read_flexowriter_symbol;
    if nas_stock = - 163 {_}
    then repeat nas_stock:= read_flexowriter_symbol
     until nas_stock <> 163
   until nas_stock <= 0
  end {word delimiter}
else if sym = 70 {>} then sym:= 71 {>=}
else if sym = 72 {=} then sym:= 80 {eqv}
else if sym = 74 {<} then sym:= 73 {<=}
else if sym = 76 {~} then sym:= 79 {imp}
else if sym = 124 {:} then sym:= 68 {div}
```

```
                  else stop(13)
                end
         else stop(14) {? or " or '}
       end;
  next_ALGOL_symbol:= sym
end {next_ALGOL_symbol};

procedure read_next_symbol;                                          {ZY}
label 1;
begin
1: case rns_state of
  ps: begin dl:= next_ALGOL_symbol;
        {store symbol in symbol store:}
        if rnsa > d7
        then begin rnsa:= rnsa div d7;
               store[rnsb]:= store[rnsb] + dl * rnsa
             end
        else begin rnsa:= d15; rnsb:= rnsb + 1; store[rnsb]:= dl * rnsa;
               if rnsb + 8 > plib then stop(25)
             end
      end;
  ms: begin {take symbol from symbol store:}
        dl:= (store[rnsd] div rnsc) mod d7;
        if rnsc > d7
        then rnsc:= rnsc div d7
        else begin rnsc:= d15; rnsd:= rnsd + 1 end
      end;
  virginal:
      begin qc:= 0; rfs_case:= 0; nas_stock:= 1;
        if scan > 0 {prescan}
        then begin rns_state:= ps;
               {initialize symbol store:}
               rnsb:= bim + 8; rnsd:= bim + 8; rnsa:= d22; rnsc:= d15;
               store[rnsb]:= 0;
             end
        else rns_state:= ms;
        goto 1
      end
  end {case}
end {read_next_symbol};

procedure read_until_next_delimiter;                                 {FT}
  label 1,3,4,5;
  var marker,elsc,bexp: integer;
```

```
  function test1: boolean;
  begin if dl = 88 {.}
    then begin dflag:= 1;
           read_next_symbol; test1:= test1
         end
    else if dl = 89 {ten} then goto 1
    else test1:= dl > 9
  end {test1};

  function test2: boolean;
  begin if dl = 89 {ten} then inw:= 1; test2:= test1
  end {test2};

  function test3: boolean;
  begin read_next_symbol; test3:= test1
  end {test3};

begin {body of read_until_next_delimiter}
  read_next_symbol;
  nflag:= 1;
  if (dl > 9) and (dl < 63) {letter}
  then begin dflag:= 0; kflag:= 0; inw:= 0;
         repeat fnw:= (inw mod d6) * d21; inw:= inw div d6 + dl * d21;
           read_next_symbol
         until (inw mod d3 > 0) or (dl > 62);
         if inw mod d3 > 0
         then begin dflag:= 1;
                fnw:= fnw + d23; marker:= 0;
                while (marker = 0) and (dl < 63) do
                begin marker:= fnw mod d6 * d21; fnw:= fnw div 64 + dl * d21;
                  read_next_symbol
                end;
                while marker = 0 do
                begin marker:= fnw mod d6 * d21;
                  fnw:= fnw div d6 + 63 * d21
                end;
                while dl < 62 do read_next_symbol
              end;
         goto 4;
       end;
  kflag:= 1; fnw:= 0; inw:= 0; dflag:= 0; elsc:= 0;
  if test2 {not (dl in [0..9,88,89])}
  then begin nflag:= 0;
         if (dl = 116 {true}) or (dl = 117 {false})
         then begin inw:= dl - 116;
```

```
                dflag:= 0; kflag:= 1; nflag:= 1;
                read_next_symbol;
                goto 4
              end;
         goto 5
       end;
  repeat if fnw < d22
    then begin inw:= 10 * inw + dl;
           fnw:= 10 * fnw + inw div d26;
           inw:= inw mod d26;
           elsc:= elsc - dflag
         end
    else elsc:= elsc - dflag + 1
  until test3;
  if (dflag = 0) and (fnw = 0)
  then goto 4;
  goto 3;
1: if test3 {not (dl in [0..9,88,89]}
  then if dl = 64 {plus}
       then begin read_next_symbol; dflag:= dl end
       else begin read_next_symbol; dflag:= - dl - 1 end
  else dflag:= dl;
  while not test3 {dl in [0..9,88,89]} do
  begin if dflag >= 0
    then dflag:= 10 * dflag + dl
    else dflag:= 10 * dflag - dl + 9;
    if abs(dflag) >= d26 then stop(3)
  end;
  if dflag < 0 then dflag:= dflag + 1;
  elsc:= elsc + dflag;
3: {float}
  if (inw = 0) and (fnw = 0)
  then begin dflag:= 0; goto 4 end;
  bexp:= 2100 {2**11 + 52; P9-characteristic};
  while fnw < d25 do
  begin inw:= 2 * inw; fnw:= 2 * fnw + inw div d26; inw:= inw mod d26;
    bexp:= bexp - 1
  end;
  if elsc > 0
  then repeat fnw:= 5 * fnw; inw:= (fnw mod 8) * d23 + (5 * inw) div 8;
         fnw:= fnw div 8;
         if fnw < d25
         then begin inw:= 2 * inw; fnw:= 2 * fnw + inw div d26;
                inw:= inw mod d26;
                bexp:= bexp - 1
```

```
              end;
         bexp:= bexp + 4; elsc:= elsc - 1;
       until elsc = 0
  else if elsc < 0
  then repeat if fnw >= 5 * d23
         then begin inw:= inw div 2 + (fnw mod 2) * d25;
                fnw:= fnw div 2; bexp:= bexp + 1
              end;
         inw:= 8 * inw; fnw:= 8 * fnw + inw div d26;
         inw:= inw mod d26 + fnw mod 5 * d26;
         fnw:= fnw div 5; inw:= inw div 5;
         bexp:= bexp - 4; elsc:= elsc + 1
       until elsc = 0;
  inw:= inw + 2048;
  if inw >= d26
  then begin inw:= 0; fnw:= fnw + 1;
         if fnw = d26 then begin fnw:= d25; bexp:= bexp + 1 end
       end;
  if (bexp < 0) or (bexp > 4095) then stop(4);
  inw:= (inw div 4096) * 4096 + bexp;
  dflag:= 1;
4: oflag:= 0;
5:
end {read_until_next_delimiter};

procedure fill_t_list(n: integer);
begin store[tlsc]:= n; tlsc:= tlsc + 1
end {fill_t_list};


procedure prescan;                                                  {HK}

  label 1,2,3,4,5,6,7;
  var bc,mbc: integer;

  procedure fill_prescan_list(n: integer); {n = 0 or n = 1}         {HF}
    var i,j,k: integer;
  begin {update plib and prescan_list chain:}
    k:= plib; plib:= k - dflag - 1; j:= k;
    for i:= 2*bc + n downto 1 do
    begin k:= store[j]; store[j]:= k - dflag - 1; j:= k end;
    {shift lower part of prescan_list down over dfag + 1 places:}
    k:= plib;
    if dflag = 0
    then for i:= j - plib downto 1 do
```

```
          begin store[k]:= store[k+1]; k:= k + 1 end
     else begin {shift:}
            for i:= j - plib - 1 downto 1 do
            begin store[k]:= store[k+2]; k:= k + 1 end;
            {enter fnw in prescan_list:}
            store[k+1]:= fnw
          end;
     {enter inw in prescan_list:}
     store[k]:= inw
   end {fill_prescan_list};

   procedure augment_prescan_list;                                {HH}
   begin dflag:= 1; inw:= plie; fnw:= plie - 1;
     fill_prescan_list(0)
   end {augment_prescan_list};

   procedure block_introduction;                                  {HK}
   begin fill_t_list(bc); fill_t_list(-1) {block-begin marker};
     mbc:= mbc + 1; bc:= mbc;
     augment_prescan_list
   end {block_introduction};

 begin {body of prescan}
   plib:= plie; store[plie]:= plie - 1; tlsc:= tlib;
   bc:= 0; mbc:= 0; qc:= 0; rht:= 0; vht:= 0;
   fill_t_list(dl); {dl should be 'begin'}
   augment_prescan_list;
 1: bflag:= 0;
 2: read_until_next_delimiter;
 3: if dl <= 84 {+,-,*,/,_:,|^,>,>=,=,<=,<,|=,~,^,`,_~,_=,goto,if,then,else}
   then {skip:} goto 1;
   if dl = 85 {for}
   then begin block_introduction; goto 1 end;
   if dl <= 89 {do,comma,period,ten} then {skip:} goto 1;
   if dl = 90 {:} then begin fill_prescan_list(0); goto 2 end;
   if dl = 91 {;}
   then begin while store[tlsc-1] < 0 {block-begin marker} do
                begin tlsc:= tlsc - 2; bc:= store[tlsc] end;
          if rht <> 0 then stop(22); if vht <> 0 then stop(23);
          goto 1
        end;
   if dl <= 97 {:=,step,until,while,comment} then {skip:} goto 1;
   if dl <= 99 {(,)}
   then begin if dl = 98 then rht:= rht + 1 else rht:= rht - 1;
          goto 1
```

```
       end;
  if dl <= 101 {[,]}
  then begin if dl = 100 then vht:= vht + 1 else vht:= vht - 1;
         goto 1
       end;
  if dl = 102 {|<}
  then begin repeat if dl = 102 {|<} then qc:= qc + 1;
               if dl = 103 {|>} then qc:= qc - 1;
               if qc > 0 then read_next_symbol
             until qc = 0;
         goto 2
       end;
  if dl = 104 {begin}
  then begin fill_t_list(dl);
         if bflag <> 0 then goto 1;
         read_until_next_delimiter;
         if (dl <= 105) or (dl > 112) then goto 3;
         tlsc:= tlsc - 1 {remove begin from t_list};
         block_introduction;
         fill_t_list(104) {add begin to t_list again};
         goto 3;
       end;
  if dl = 105 {end}
  then begin while store[tlsc-1] < 0 {block-begin marker} do
             begin tlsc:= tlsc - 2; bc:= store[tlsc] end;
         if rht <> 0 then stop(22); if vht <> 0 then stop(23);
         tlsc:= tlsc - 1 {remove corresponding begin from t_list};
         if tlsc > tlib then goto 1;
         goto 7 {end of prescan}
       end;
  if dl <= 105 {dl = |>} then goto 1;
  if dl = 111 {switch}
  then if bflag = 0
       then {declarator}
            begin read_until_next_delimiter {for switch identifier};
              fill_prescan_list(0); goto 6
            end
       else {specifier}
            goto 5;
4: if dl = 112 {procedure}
   then if bflag = 0
        then {declarator}
             begin bflag:= 1;
               read_until_next_delimiter {for procedure identifier};
               fill_prescan_list(1); block_introduction; goto 6
```

```
             end
        else {specificier}
             goto 5;
   if dl > 117 {false} then stop(8);
5: read_until_next_delimiter;
6: if dl <> 91 {;} then goto 4;
   goto 2;
7:
end {prescan};

procedure intro_new_block2;                                         {HW}
label 1;
var i,w: integer;
begin inba:= d17 + d15;
1: i:= plib; plib:= store[i]; i:= i + 1;
  while i <> plib do
  begin w:= store[i];
    if w mod 8 = 0 {at most 4 letters/digits}
    then i:= i + 1
    else begin store[nlib+nlsc]:=store[i+1]; i:= i + 2; nlsc:= nlsc + 1 end;
    store[nlib+nlsc]:= w; nlsc:= nlsc + 2;
    if nlib + nlsc > i then stop(15);
    store[nlib+nlsc-1]:= bn * d19 + inba
  end;
  if inba <> d18 + d15
  then begin inba:= d18 + d15; goto 1 end;
  lvc:= 0
end {intro_new_block2};

procedure intro_new_block1;                                         {HW}
begin fill_t_list(nlsc); fill_t_list(161);
  intro_new_block2
end {intro_new_block1};

procedure intro_new_block;                                          {HW}
begin bn:= bn + 1; intro_new_block1
end {intro_new_block};

procedure bit_string_maker(w: integer);                             {LL}
var head,tail,i: integer;
begin head:= 0; tail:= w mod d10;
  {shift (head,tail) bitcount places to the left:}
  for i:= 1 to bitcount do
  begin head:= 2 * head + tail div d26; tail:= (tail mod d26) * 2
```

```
  end {shift};
  bitstock:= bitstock + tail; bitcount:= bitcount + w div d10;
  if bitcount > 27
  then begin bitcount:= bitcount - 27;
         store[rnsb]:= bitstock; bitstock:= head; rnsb:= rnsb + 1;
         if rnsb = rnsd
         then if nlib + nlsc + 8 < plib
              then begin {shift text, fli, kli and nli}
                     for i:= nlib + nlsc - rnsd - 1 downto 0 do
                     store[rnsd+i+8]:= store[rnsd+i];
                     rnsd:= rnsd + 8; flib:= flib + 8;
                     klib:= klib + 8; nlib:= nlib + 8
                   end
              else stop(25)
       end
end {bit_string_maker};

procedure address_coder(a: integer);                                {LS}
var w: integer;
begin w:= a mod d5;
  if w = 1 then w:= 2048 {2*1024 +  0} else
  if w = 2 then w:= 3074 {3*1024 +  2} else
  if w = 3 then w:= 3075 {3*1024 +  3}
           else w:= 6176 {6*1024 + 32} + w;
  bit_string_maker(w);
  w:= (a div d5) mod d5;
  if w = 0 then w:= 2048 {2*1024 +  0} else
  if w = 1 then w:= 4100 {4*1024 +  4} else
  if w = 2 then w:= 4101 {4*1024 +  5} else
  if w = 4 then w:= 4102 {4*1024 +  6} else
  if w = 5 then w:= 4103 {4*1024 +  7}
           else w:= 6176 {6*1024 + 32} + w;
  bit_string_maker(w);
  w:= (a div d10) mod d5;
  if w = 0 then w:= 1024 {1*1024 + 0}
           else w:= 6176 {6*1024 + 32} + w;
  bit_string_maker(w)
end {address_coder};

procedure fill_result_list(opc,w: integer);                         {ZF}
var j: 8..61;
begin rlsc:= rlsc + 1;
  if opc < 8
  then begin address_coder(w);
         w:= (w div d15) * d15 + opc;
```

```
       if w = 21495808 {  2S   0 A  } then w:= 3076 {3*1024 +   4} else
       if w = 71827459 {  2B   3 A  } then w:= 3077 {3*1024 +   5} else
       if w = 88080386 {  2T 2X0    } then w:= 4108 {4*1024 +  12} else
       if w = 71827456 {  2B   0 A  } then w:= 4109 {4*1024 +  13} else
       if w =  4718592 {  2A   0 A  } then w:= 7280 {7*1024 + 112} else
       if w = 71303170 {  2B 2X0    } then w:= 7281 {7*1024 + 113} else
       if w = 88604673 {  2T   1 A  } then w:= 7282 {7*1024 + 114} else
       if w =        0 {  0A 0X0    } then w:= 7283 {7*1024 + 115} else
       if w =   524291 {  0A   3 A  } then w:= 7284 {7*1024 + 116} else
       if w = 88178690 {N 2T 2X0    } then w:= 7285 {7*1024 + 117} else
       if w = 71827457 {  2B   1 A  } then w:= 7286 {7*1024 + 118} else
       if w =  1048577 {  0A 1X0 B  } then w:= 7287 {7*1024 + 119} else
       if w = 20971522 {  2S 2X0    } then w:= 7288 {7*1024 + 120} else
       if w =  4784128 {Y 2A   0 A  } then w:= 7289 {7*1024 + 121} else
       if w =  8388608 {  4A 0X0    } then w:= 7290 {7*1024 + 122} else
       if w =  4390912 {Y 2A 0X0   P} then w:= 7291 {7*1024 + 123} else
       if w = 13172736 {Y 6A   0 A  } then w:= 7292 {7*1024 + 124} else
       if w =  1572865 {  0A 1X0 C  } then w:= 7293 {7*1024 + 125} else
       if w =   524288 {  0A   0 A  } then w:= 7294 {7*1024 + 126}
       else begin address_coder(w div d15 + opc * d12);
              w:= 7295 {7*1024 + 127}
            end
     end {opc < 8}
else if opc <= 61
then begin j:= opc;
       case j of
          8: w:= 10624 {10*1024+384};  9: w:=  6160 { 6*1024+ 16};
         10: w:= 10625 {10*1024+385}; 11: w:= 10626 {10*1024+386};
         12: w:= 10627 {10*1024+387}; 13: w:=  7208 { 7*1024+ 40};
         14: w:=  6161 { 6*1024+ 17}; 15: w:= 10628 {10*1024+388};
         16: w:=  5124 { 5*1024+  4}; 17: w:=  7209 { 7*1024+ 41};
         18: w:=  6162 { 6*1024+ 18}; 19: w:=  7210 { 7*1024+ 42};
         20: w:=  7211 { 7*1024+ 43}; 21: w:= 10629 {10*1024+389};
         22: w:= 10630 {10*1024+390}; 23: w:= 10631 {10*1024+391};
         24: w:= 10632 {10*1024+392}; 25: w:= 10633 {10*1024+393};
         26: w:= 10634 {10*1024+394}; 27: w:= 10635 {10*1024+395};
         28: w:= 10636 {10*1024+396}; 29: w:= 10637 {10*1024+397};
         30: w:=  6163 { 6*1024+ 19}; 31: w:=  7212 { 7*1024+ 44};
         32: w:= 10638 {10*1024+398}; 33: w:=  4096 { 4*1024+  0};
         34: w:=  4097 { 4*1024+  1}; 35: w:=  7213 { 7*1024+ 45};
         36: w:= 10639 {10*1024+399}; 37: w:= 10640 {10*1024+400};
         38: w:= 10641 {10*1024+401}; 39: w:=  7214 { 7*1024+ 46};
         40: w:= 10642 {10*1024+402}; 41: w:= 10643 {10*1024+403};
         42: w:= 10644 {10*1024+404}; 43: w:= 10645 {10*1024+405};
         44: w:= 10646 {10*1024+406}; 45: w:= 10647 {10*1024+407};
```

```
           46: w:= 10648 {10*1024+408}; 47: w:= 10649 {10*1024+409};
           48: w:= 10650 {10*1024+410}; 49: w:= 10651 {10*1024+411};
           50: w:= 10652 {10*1024+412}; 51: w:= 10653 {10*1024+413};
           52: w:= 10654 {10*1024+414}; 53: w:= 10655 {10*1024+415};
           54: w:= 10656 {10*1024+416}; 55: w:= 10657 {10*1024+417};
           56: w:=  5125 { 5*1024+  5}; 57: w:= 10658 {10*1024+418};
           58: w:=  5126 { 5*1024+  6}; 59: w:= 10659 {10*1024+419};
           60: w:= 10660 {10*1024+420}; 61: w:=  7215 { 7*1024+ 47}
         end {case}
       end {opc <= 61}
  else if opc = 85{ST}
  then w:=  5127 { 5*1024 +   7}
  else w:= 10599 {10*1024 + 359} + opc;
  bit_string_maker(w)
end {fill_result_list};


procedure main_scan;                                                {EL}

  label 1,2,3,64,66,69,70,76,81,82,8201,8202,83,8301,84,8401,85,8501,
        86,8601,87,8701,8702,8703,8704,8705,
        90,91,92,94,95,96,98,9801,9802,9803,9804,99,100,101,
        102,104,105,1052,106,107,108,1081,1082,1083,
        109,110,1101,1102,1103,111,112,1121,1122,1123,1124;

  procedure fill_t_list_with_delimiter;                             {ZW}
  begin fill_t_list(d8*oh+dl)
  end {fill_t_list_with_delimiter};

  procedure fill_future_list(place,value: integer);                 {FU}
  var i: integer;
  begin if place >= klib
    then begin if nlib + nlsc + 16 >= plib then stop(6);
           for i:= nlib + nlsc - 1 downto klib do
           store[i+16]:= store[i];
           klib:= klib + 16; nlib:= nlib + 16
         end;
    store[place]:= value
  end {fill_future_list};

  procedure fill_constant_list(n: integer);                         {KU}
  var i: integer;
  begin if klib + klsc = nlib
    then begin if nlib + nlsc + 16 >= plib then stop(18);
           for i:= nlib + nlsc - 1 downto nlib do
```

```
         store[i+16]:= store[i];
         nlib:= nlib + 16
       end;
  if n >= 0
  then store[klib+klsc]:= n
  else {one’s complement representation} store[klib+klsc]:= mz + n;
  klsc:= klsc + 1
end {fill_constant_list};

procedure unload_t_list_element(var variable: integer);           {ZU}
begin tlsc:= tlsc - 1; variable:= store[tlsc]
end {unload_t_list_element};

procedure fill_output(c: integer);
begin pos:= pos + 1;
  if c < 10 then write(chr(c+ord(’0’)))
  else if c < 36 then write(chr(c-10+ord(’a’)))
  else if c < 64 then write(chr(c-37+ord(’A’)))
  else if c = 184 then write(’ ’)
  else if c = 138
       then begin write(’ ’:8 - (pos - 1) mod 8);
              pos:= pos + 8 - (pos - 1) mod 8
            end
  else begin writeln; pos:= 0 end
end {fill_output};

procedure offer_character_to_typewriter(c: integer);              {HS}
begin c:= c mod 64;
  if c < 63 then fill_output(c)
end {offer_character_to_typewriter};

procedure label_declaration;                                      {FY}
var id,id2,i,w: integer;
begin id:= store[nlib+nid];
  if (id div d15) mod 2 = 0
  then begin {preceding applied occurrences}
         fill_future_list(flib+id mod d15,rlsc)
       end
  else {first occurrence}
       store[nlib+nid]:= id - d15 + 1 * d24 + rlsc;
  id:= store[nlib+nid-1];
  if id mod d3 = 0
  then begin {at most 4 letters/digits}
         i:= 4; id:= id div d3;
         while (id mod d6) = 0{void} do
```

```
          begin i:= i - 1; id:= id div d6 end;
          repeat offer_character_to_typewriter(id);
            i:= i - 1; id:= id div d6
          until i = 0
        end
  else begin id2:= store[nlib+nid-2];
          id2:= id2 div d3 + (id2 mod d3) * d24;
          w:= (id2 mod d24) * d3 + id div d24;
          id:= (id mod d24) * d3 + id2 div d24;
          id2:= w;
          i:= 9;
          repeat offer_character_to_typewriter(id);
            i:= i - 1;
            w:= id2 div d6 + (id mod d6) * d21;
            id:= id div d6 + (id2 mod d6) * d21;
            id2:= w
          until i = 0
        end;
  fill_output(138{TAB});
  w:= rlsc;
  for i:= 1 to 3 do
  begin offer_character_to_typewriter(w div d10 div 10);
    offer_character_to_typewriter(w div d10 mod 10);
    w:= (w mod d10) * d5;
    if i < 3 then fill_output(184{SPACE})
  end;
  fill_output(139{NLCR})
end {label_declaration};

procedure test_first_occurrence;                                  {LF}
begin id:= store[nlib+nid];
  if (id div d15) mod 2 = 1 {first occurrence}
  then begin id:= id - d15 - id mod d15 + 2 * d24 + flsc;
          if nid <= nlsc0 {MCP}
          then fill_future_list(flib+flsc,store[nlib+nid]);
          store[nlib+nid]:= id;
          flsc:= flsc + 1
        end
end {test_first_occurrence};

procedure new_block_by_declaration1;                              {HU}
begin fill_result_list(0,71827456+bn) {2B 'bn' A};
  fill_result_list(89{SCC},0);
  pnlv:= 5 * 32 + bn; vlam:= pnlv
end {new_block_by_declaration1};
```

```
procedure new_block_by_declaration;                               {HU}
begin if store[tlsc-2] <> 161{block-begin marker}
  then begin tlsc:= tlsc - 1 {remove 'begin'};
         fill_result_list(0,4718592) {2A 0 A};
         fill_result_list(1,71827456+rlsc+3) {2B 'rlsc+3' A};
         fill_result_list(9{ETMP},0);
         fill_result_list(2,88080384+flsc) {2T 'flsc'};
         fill_t_list(flsc); flsc:= flsc + 1;
         intro_new_block;
         fill_t_list(104{begin});
         new_block_by_declaration1
       end
end {new_block_by_declaration};

procedure fill_name_list;                                         {HN}
begin nlsc:= nlsc + dflag + 2;
  if nlsc + nlib > plib then stop(16);
  store[nlib+nlsc-1]:= id; store[nlib+nlsc-2]:= inw;
  if inw mod d3 > 0 then store[nlib+nlsc-3]:= fnw
end {fill_name_list};

procedure reservation_of_local_variables;                         {KY}
begin if lvc > 0
  then begin fill_result_list(0,4718592+lvc) {2A 'lvc' A};
         fill_result_list(0,8388657) {4A 17X1};
         fill_result_list(0,8388658) {4A 18X1}
       end
end {reservation_of_local_variables};

procedure address_to_register;                                    {ZR}
begin if id div d15 mod 2 = 0 {static addressing}
  then if id div d24 mod d2 = 2 {future list}
       then fill_result_list(2,
              71303168+id mod d15{2B 'FLI-address'})
       else fill_result_list(id div d24 mod 4,
              71827456+id mod d15{2B 'static address' A})
  else fill_result_list(0,
              21495808+id mod d15{2S 'dynamic address' A})
end {address_to_register};

procedure generate_address;                                       {ZH}
var opc: integer;
begin address_to_register;
  if (id div d16) mod 2 = 1
```

```
  then {formal} fill_result_list(18{TFA},0)
  else begin opc:= 14{TRAD};
         if (id div d15) mod 2 = 0 then opc:= opc + 1{TRAS};
         if (id div d19) mod 2 = 1 then opc:= opc + 2{TIAD or TIAS};
         fill_result_list(opc,0)
       end
end {generate_address};

procedure reservation_of_arrays;                                   {KN}
begin if vlam <> 0
  then begin vlam:= 0;
         if store[tlsc-1] = 161{block-begin marker}
         then rlaa:= nlib + store[tlsc-2]
         else rlaa:= nlib + store[tlsc-3];
         rlab:= nlib + nlsc;
         while rlab <> rlaa do
         begin id:= store[rlab-1];
           if (id >= d26) and (id < d25 + d26)
           then begin {value array:}
                  address_to_register;
                  if (id div d19) mod 2 = 0
                  then fill_result_list(92{RVA},0)
                  else fill_result_list(93{IVA},0);
                  store[rlab-1]:= (id div d15) * d15 - d16 + pnlv;
                  pnlv:= pnlv + 8 * 32 {at most 5 indices}
                end;
           if store[rlab-2] mod d3 = 0
           then rlab:= rlab - 2 else rlab:= rlab - 3
         end;
         rlab:= nlib + nlsc;
         while rlab <> rlaa do
         begin if store[rlab-1] >= d26
           then begin id:= store[rlab-1] - d26;
                  if id < d25
                  then begin address_to_register;
                         fill_result_list(95{VAP},0)
                       end
                  else begin id:= id - d25;
                         address_to_register;
                         fill_result_list(94{LAP},0)
                       end
                end;
           if store[rlab-2] mod d3 = 0
           then rlab:= rlab - 2 else rlab:= rlab - 3
         end;
```

```
             if nflag <> 0
             then id:= store[nlib+nid]
           end
    end {reservation_of_arrays};

    procedure procedure_statement;                                 {LH}
    begin if eflag = 0 then reservation_of_arrays;
      if nid > nlscop
      then begin if fflag = 0 then test_first_occurrence;
             address_to_register
           end
      else begin fill_t_list(store[nlib+nid] mod d12);
             if dl = 98{(}
             then begin eflag:= 1; goto 9801 end
           end
    end {procedure_statement};

    procedure production_transmark;                                {ZL}
    begin fill_result_list(9+2*fflag-eflag,0)
    end {production_transmark};

    procedure production_of_object_program(opht: integer);         {ZS}
    var operator,block_number: integer;
    begin oh:= opht;
      if nflag <> 0
      then begin nflag:= 0; aflag:= 0;
             if pflag = 0
             then if jflag = 0
                  then begin address_to_register;
                         if oh > (store[tlsc-1] div d8) mod 16
                         then operator:= 315{5*63}
                         else begin operator:= store[tlsc-1] mod d8;
                                if (operator <= 63) or (operator > 67)
                                then operator:= 315{5*63}
                                else begin tlsc:= tlsc - 1;
                                       operator:= 5 * operator
                                     end
                              end;
                         if fflag = 0
                         then begin if id div d15 mod 2 = 0
                                then operator:= operator + 1;
                                if id div d19 mod 2 <> 0
                                then operator:= operator + 2;
                                fill_result_list(operator-284,0)
                              end
```

```
                   else fill_result_list(operator-280,0)
                 end
            else if fflag = 0
                 then begin block_number:= id div d19 mod d5;
                        if block_number <> bn
                        then begin fill_result_list
                                     (0,71827456+block_number);
                               fill_result_list(28{GTA},0)
                             end;
                        test_first_occurrence;
                        if id div d24 mod 4 = 2
                        then fill_result_list(2,88080384+id mod d15)
                             {2T 'address'}
                        else fill_result_list(1,88604672+id mod d15)
                             {2T 'address' A}
                      end
                 else begin address_to_register;
                        fill_result_list(35{TFR},0)
                      end
       else begin procedure_statement;
              if nid > nlscop
              then begin fill_result_list(0,4718592{2A 0 A});
                     production_transmark
                   end
            end
     end
else if aflag <> 0
then begin aflag:= 0; fill_result_list(58{TAR},0) end;
while oh <= store[tlsc-1] div d8 mod 16 do
begin tlsc:= tlsc - 1; operator:= store[tlsc] mod d8;
  if (operator > 63) and (operator<= 80)
  then fill_result_list(operator-5,0)
  else if operator = 132 {NEG}
  then fill_result_list(57{NEG},0)
  else if (operator < 132) and (operator > 127)
  then begin {ST,STA,STP,STAP}
         if operator > 129
         then begin {STP,STAP}
                tlsc:= tlsc - 1;
                fill_result_list(0,71827456+store[tlsc]{2B 'BN' A})
              end;
         fill_result_list(operator-43,0)
       end
  else {special function}
  if (operator > 127) and (operator <= 141)
```

```
      then fill_result_list(operator-57,0)
      else if (operator > 141) and (operator <= 151)
      then fill_result_list(operator-40,0)
      else stop(22)
    end
  end {production_of_object_program};

  function  thenelse: boolean;                                {ZN}
  begin if (store[tlsc-1] mod 255 = 83{then})
        or (store[tlsc-1] mod 255 = 84{else})
    then begin tlsc:= tlsc - 2;
           fill_future_list(flib+store[tlsc],rlsc);
           unload_t_list_element(eflag);
           thenelse:= true
         end
    else thenelse:= false
  end {thenelse};

  procedure empty_t_list_through_thenelse;                    {FR}
  begin oflag:= 1;
    repeat production_of_object_program(1)
    until not thenelse
  end {empty_t_list_through_thenelse};

  function do_in_t_list: boolean;                             {ER}
  begin if store[tlsc-1] mod 255 = 86
   then begin tlsc:= tlsc - 5;
          nlsc:= store[tlsc+2]; bn:= bn - 1;
          fill_future_list(flib+store[tlsc+1],rlsc+1);
          fill_result_list(1,88604672{2T 0X0 A}+store[tlsc]);
          do_in_t_list:= true
        end
   else do_in_t_list:= false
  end {do_in_t_list};

  procedure look_for_name;                                    {HZ}
  label 1,2;
  var i,w: integer;
  begin i:= nlib + nlsc;
  1: w:= store[i-2];
    if w = inw
    then if w mod 8 = 0
         then {at most 4 letters/digits} goto 2
         else {more than 4 letters/digits}
              if store[i-3] = fnw then goto 2;
```

```
    if w mod 8 = 0 then i:= i - 2 else i:= i - 3;
    if i > nlib then goto 1;
    stop(7);
  2: nid:= i - nlib - 1; id:= store[i-1];
    pflag:= id div d18 mod 2;
    jflag:= id div d17 mod 2;
    fflag:= id div d16 mod 2
  end {look_for_name};

  procedure look_for_constant;                                    {FW}
  var i: integer;
  begin if klib + klsc + dflag >= nlib
    then begin {move name list}
           if nlib + nlsc + 16 >= plib then stop(5);
           for i:= nlsc - 1 downto 0 do
             store[nlib+i+16]:= store[nlib+i];
           nlib:= nlib + 16
         end;
    if dflag = 0
    then begin {search integer constant}
           store[klib+klsc]:= inw;
           i:= 0;
           while store[klib+i] <> inw do i:= i + 1
         end
    else begin {search floating constant}
           store[klib+klsc]:= fnw; store[klib+klsc+1]:= inw;
           i:= 0;
           while (store[klib+i] <> fnw)
             or (store[klib+i+1] <> inw) do i:= i + 1
         end;
    if i = klsc
    then {first occurrence} klsc:= klsc + dflag + 1;
    id:= 3 * d24 + i;
    if dflag = 0 then id:= id + d19;
    jflag:= 0; pflag:= 0; fflag:= 0
  end {look_for_constant};

begin {body of main scan}                                         {EL}
  1: read_until_next_delimiter;
  2: if nflag <> 0
     then if kflag = 0
          then look_for_name
          else look_for_constant
     else begin jflag:= 0; pflag:= 0; fflag:= 0 end;
  3: if dl <= 65 then goto 64; {+,-}                              {EH}
```

```
    if dl <= 68 then goto 66; {*,/,_:}
    if dl <= 69 then goto 69; {|^}
    if dl <= 75 then goto 70; {<,_<,=,_>,>,|=}
    if dl <= 80 then goto 76; {˜,ˆ,‘,=>,_=}
    case dl of
     81: goto  81; {goto}                                         {KR}
     82: goto  82; {if}                                           {EY}
     83: goto  83; {then}                                         {EN}
     84: goto  84; {else}                                         {FZ}
     85: goto  85; {for}                                          {FE}
     86: goto  86; {do}                                           {FL}
     87: goto  87; {,}                                            {EK}
     90: goto  90; {:}                                            {FN}
     91: goto  91; {;}                                            {FS}
     92: goto  92; {:=}                                           {EZ}
     94: goto  94; {step}                                         {FH}
     95: goto  95; {until}                                        {FK}
     96: goto  96; {while}                                        {FF}
     98: goto  98; {(}                                            {EW}
     99: goto  99; {)}                                            {EU}
    100: goto 100; {[}                                            {EE}
    101: goto 101; {]}                                            {EF}
    102: goto 102; {|<}                                           {KS}
    104: goto 104; {begin}                                        {LZ}
    105: goto 105; {end}                                          {FS}
    106: goto 106; {own}                                          {KH}
    107: goto 107; {Boolean}                                      {KZ}
    108: goto 108; {integer}                                      {KZ}
    109: goto 109; {real}                                         {KE}
    110: goto 110; {array}                                        {KF}
    111: goto 111; {switch}                                       {HE}
    112: goto 112; {procedure}                                    {HY}
    end {case};

64: {+,-}                                                         {ES}
    if oflag = 0
    then begin production_of_object_program(9);
           fill_t_list_with_delimiter
        end
    else if dl = 65{-}
         then begin oh:= 10; dl:= 132{NEG};
                fill_t_list_with_delimiter
              end;
    goto 1;
```

```
66: {*,/,_:}                                                  {ET}
    production_of_object_program(10);
    fill_t_list_with_delimiter;
    goto 1;

69: {|^}                                                      {KT}
    production_of_object_program(11);
    fill_t_list_with_delimiter;
    goto 1;

70: {<,_<,=,_>,>,|=}                                          {KK}
    oflag:= 1;
    production_of_object_program(8);
    fill_t_list_with_delimiter;
    goto 1;

76: {~,^,‘,=>,_=}                                             {KL}
    if dl = 76{~}
    then begin oh:= 83-dl; goto 8202 end;
    production_of_object_program(83-dl);
    fill_t_list_with_delimiter;
    goto 1;

81: {goto}                                                    {KR}
    reservation_of_arrays; goto 1;

82:   {if}                                                    {EY}
      if eflag = 0 then reservation_of_arrays;
      fill_t_list(eflag); eflag:= 1;
8201: oh:= 0;
8202: fill_t_list_with_delimiter;
      oflag:= 1; goto 1;

83:   {then}                                                  {EN}
      repeat production_of_object_program(1) until not thenelse;
      tlsc:= tlsc - 1; eflag:= store[tlsc-1];
      fill_result_list(30{CAC},0);
      fill_result_list(2,88178688+flsc) {N 2T 'flsc'};
8301: fill_t_list(flsc); flsc:= flsc + 1;
      goto 8201;

84:   {else}                                                  {FZ}
      production_of_object_program(1);
      if store[tlsc-1] mod d8 = 84{else}
      then if thenelse then goto 84;
```

```
8401: if do_in_t_list then goto 8401;
      if store[tlsc-1] = 161 {block-begin marker}
      then begin tlsc:= tlsc - 3;
             nlsc:= store[tlsc+1];
             fill_future_list(flib+store[tlsc],rlsc+1);
             fill_result_list(12{RET},0);
             bn:= bn - 1; goto 8401
           end;
      fill_result_list(2,88080384+flsc) {2T 'flsc'};
      if thenelse {finds 'then'!}
      then tlsc:= tlsc + 1 {keep eflag in t_list};
      goto 8301;

85:   {for}                                                 {FE}
      reservation_of_arrays;
      fill_result_list(2,88080384+flsc) {2T 'flsc'};
      fora:= flsc; flsc:= flsc + 1;
      fill_t_list(rlsc);
      vflag:= 1; bn:= bn + 1;
8501: oh:= 0; fill_t_list_with_delimiter;
      goto 1;

86:   {do}                                                  {FL}
      empty_t_list_through_thenelse;
      goto 8701; {execute part of DDEL ,}
8601: {returned from DDEL ,}
     vflag:= 0; tlsc:= tlsc - 1;
     fill_result_list(2,20971520+flsc) {2S 'flsc'};
     fill_t_list(flsc); flsc:= flsc + 1;
     fill_result_list(27{FOR8},0);
     fill_future_list(flib+fora,rlsc);
     fill_result_list(19{FOR0},0);
     fill_result_list(1,88604672{2T 0X0 A}+store[tlsc-2]);
     fill_future_list(flib+forc,rlsc);
     eflag:= 0; intro_new_block1;
     goto 8501;

87:  {,}                                                    {EK}
     oflag:= 1;
     if iflag = 1
     then begin {subscript separator:}
            repeat production_of_object_program(1)
            until not thenelse;
            goto 1
          end;
```

```
       if vflag = 0 then goto 8702;
       {for-list separator:}
       repeat production_of_object_program(1)
       until not thenelse;
8701:  if store[tlsc-1] mod d8 = 85{for}
       then fill_result_list(21{for2},0)
       else begin tlsc:= tlsc - 1;
              if store[tlsc] mod d8 = 96{while}
              then fill_result_list(23{for4},0)
              else fill_result_list(26{for7},0)
            end;
       if dl = 86{do} then goto 8601;
       goto 1;
8702:  if mflag = 0 then goto 8705;
       {actual parameter separator:}
       if store[tlsc-1] mod d8 = 87{,}
       then if aflag = 0
            then if (store[tlsc-2] = rlsc)
                    and (fflag = 0) and (jflag = 0) and (nflag = 1)
                 then begin if nid > nlscop
                        then begin if (pflag = 1) and (fflag = 0)
                               then {non-formal procedure:}
                                    test_first_occurrence;
                               {PORD construction:}
                               if (id div d15) mod 2 = 0
                               then begin {static addressing}
                                      pstb:= ((id div d24) mod d2) * d24
                                             + id mod d15;
                                      if (id div d24) mod d2 = 2
                                      then pstb:= pstb + d17
                                    end
                               else begin{dynamic addressing}
                                      pstb:= d16 + (id mod d5) * d22
                                             + (id div d5) mod d10;
                                      if (id div d16) mod 2 = 1
                                      then begin store[tlsc-2]:= pstb + d17;
                                             goto 8704
                                           end
                                    end;
                               if (id div d18) mod 2 = 1
                               then store[tlsc-2]:= pstb + d20
                               else if (id div d19) mod 2 = 1
                               then store[tlsc-2]:= pstb + d19
                               else store[tlsc-2]:= pstb;
                               goto 8704
```

```
                            end
                       else begin fill_result_list(98{TFP},0);
                              goto 8703
                            end
                     end
                else goto 8703
           else begin {completion of implicit subroutine:}
                 store[tlsc-2]:= store[tlsc-2] + d19 + d20 + d24;
                 fill_result_list(13{EIS},0); goto 8704
                end;
8703: {completion of implicit subroutine:}
      repeat production_of_object_program(1)
      until not (thenelse or do_in_t_list);
      store[tlsc-2]:= store[tlsc-2] + d20 + d24;
      fill_result_list(13{EIS},0);
8704: if dl = 87{,} then goto 9804 {prepare next parameter};
      {production of PORDs:}
      psta:= 0; unload_t_list_element(pstb);
      while pstb mod d8 = 87{,} do
      begin psta:= psta + 1; unload_t_list_element(pstb);
        if pstb div d16 mod 2 = 0
        then fill_result_list(pstb div d24, pstb mod d24)
        else fill_result_list(0,pstb);
        unload_t_list_element(pstb)
      end;
      tlsc:= tlsc - 1;
      fill_future_list(flib+store[tlsc],rlsc);
      fill_result_list(0,4718592+psta) {2A 'psta' A};
      bn:= bn - 1;
      unload_t_list_element(fflag); unload_t_list_element(eflag);
      production_transmark;
      aflag:= 0;
      unload_t_list_element(mflag); unload_t_list_element(vflag);
      unload_t_list_element(iflag); goto 1;
8705: empty_t_list_through_thenelse;
      if sflag = 0 then {array declaration} goto 1;
      {switch declaration:}
      oh:= 0; dl:= 160;
      fill_t_list(rlsc); fill_t_list_with_delimiter; goto 1;

 90: {:}                                                           {FN}
     if jflag = 0
     then begin {array declaration}
            ic:= ic + 1;
            empty_t_list_through_thenelse
```

```
         end
    else begin {label declaration}
           reservation_of_arrays;
           label_declaration
         end;
    goto 1;

91: goto 105{end};

92: {:=}                                                          {EZ}
    reservation_of_arrays;
    dl:= 128{ST}; oflag:= 1;
    if vflag = 0
    then begin if sflag = 0
           then begin {assignment statement}
                  if eflag = 0
                  then eflag:= 1
                  else dl:= 129{STA};
                  oh:= 2;
                  if pflag = 0
                  then begin {assignment to variable}
                         if nflag <> 0
                         then {assignment to scalar} generate_address;
                       end
                  else begin {assignment to function identifier}
                         dl:= dl + 2{STP or STAP};
                         fill_t_list((id div d19) mod d5{bn from id})
                       end;
                  fill_t_list_with_delimiter
                end
           else begin {switch declaration}
                  fill_result_list(2,88080384+flsc) {2T 'flsc'};
                  fill_t_list(flsc); flsc:= flsc + 1;
                  fill_t_list(nid);
                  oh:= 0; fill_t_list_with_delimiter;
                  dl:= 160;
                  fill_t_list(rlsc); fill_t_list_with_delimiter
                end
         end
    else begin {for statement}
           eflag:= 1;
           if nflag <> 0 then {simple variable} generate_address;
           fill_result_list(20{FOR1},0);
           forc:= flsc;
           fill_result_list(2,88080384+flsc) {2T 'flsc'};
```

```
            flsc:= flsc + 1;
            fill_future_list(flib+fora,rlsc);
            fill_result_list(0,4718592{2A 0 A});
            fora:= flsc;
            fill_result_list(2,71303168+flsc) {2B 'flsc};
            flsc:= flsc + 1;
            fill_result_list(9{ETMP},0)
          end;
       goto 1;

94: {step}                                                  {FH}
     empty_t_list_through_thenelse;
     fill_result_list(24{FOR5},0);
     goto 1;

95: {until}                                                 {FK}
     empty_t_list_through_thenelse;
     fill_result_list(25{FOR6},0);
     goto 8501;

96: {while}                                                 {FF}
     empty_t_list_through_thenelse;
     fill_result_list(22{FOR3},0);
     goto 8501;

98:   {(}                                                   {EW}
      oflag:= 1;
      if pflag = 1 then goto 9803;
9801: {parenthesis in expression:}
      fill_t_list(mflag);
      mflag:= 0;
9802: oh:= 0; fill_t_list_with_delimiter;
      goto 1;
9803: {begin of parameter list:}
      procedure_statement;
      fill_result_list(2,88080384+flsc) {2T 'flsc'};
      fill_t_list(iflag); fill_t_list(vflag);
      fill_t_list(mflag); fill_t_list(eflag);
      fill_t_list(fflag); fill_t_list(flsc);
      iflag:= 0; vflag:= 0; mflag:= 1; eflag:= 1;
      flsc:= flsc + 1; oh:= 0; bn:= bn + 1;
      fill_t_list_with_delimiter;
      dl:= 87{,};
9804: {prepare parsing of actual parameter:}
      fill_t_list(rlsc);
```

```
        aflag:= 0; goto 9802;

 99: {()}                                                      {EU}
     if mflag = 1 then goto 8702;
     repeat production_of_object_program(1)
     until not thenelse;
     tlsc:= tlsc - 1; unload_t_list_element(mflag);
     goto 1;

100: {[}                                                       {EE}
     if eflag = 0 then reservation_of_arrays;
     oflag:= 1; oh:= 0;
     fill_t_list(eflag); fill_t_list(iflag);
     fill_t_list(mflag); fill_t_list(fflag);
     fill_t_list(jflag); fill_t_list(nid);
     eflag:= 1; iflag:= 1; mflag:= 0;
     fill_t_list_with_delimiter;
     if jflag = 0 then generate_address {of storage function};
     goto 1;

101: {]}                                                       {EF}
     repeat production_of_object_program(1)
     until not thenelse;
     tlsc:= tlsc - 1;
     if iflag = 0
     then begin {array declaration:}
            fill_result_list(0,21495808+aic{2S 'aic' A});
            fill_result_list(90{RSF}+ibd,0) {RSF or ISF};
            arrb:= d15 + d25 + d26;
            if ibd = 1 then arrb:= arrb + d19;
            arra:= nlib + nlsc;
            repeat store[arra-1]:= arrb + pnlv;
              if store[arra-2] mod d3 = 0
              then arra:= arra - 2 else arra:= arra - 3;
              pnlv:= pnlv + (ic + 3) * d5; aic:= aic - 1
            until aic = 0;
            read_until_next_delimiter;
            if dl <> 91 then goto 1103;
            eflag:= 0; goto 1
          end;
     unload_t_list_element(nid); unload_t_list_element(jflag);
     unload_t_list_element(fflag); unload_t_list_element(mflag);
     unload_t_list_element(iflag); unload_t_list_element(eflag);
     if jflag = 0
     then begin {subscripted variable:}
```

```
            aflag:= 1; fill_result_list(56{IND},0);
            goto 1
          end;
     {switch designator:}
     nflag:= 1; fill_result_list(29{SSI},0);
     read_next_symbol;
     id:= store[nlib+nid];
     pflag:= 0; goto 3;

102: {|<}                                                           {KS}
     qc:= 1; qb:= 0; qa:= 1;
     repeat read_next_symbol;
       if dl = 102{|<} then qc:= qc + 1;
       if dl = 103{|>} then qc:= qc - 1;
       if qc > 0
       then begin qb:= qb + dl * qa; qa:= qa * d8;
              if qa = d24
              then begin fill_result_list(0,qb); qb:= 0; qa:= 1 end
            end
     until qc = 0;
     fill_result_list(0,qb+255{end marker}*qa);
     oflag:= 0; goto 1;

104: {begin}                                                        {LZ}
     if store[tlsc-1] <> 161 {block-begin marker}
     then reservation_of_arrays;
     goto 8501;

105: {end}                                                          {FS}
     reservation_of_arrays;
     repeat empty_t_list_through_thenelse
     until not do_in_t_list;
     if sflag = 0
     then begin if store[tlsc-1] = 161 {blok-begin marker}
            then begin tlsc:= tlsc - 3;
                   nlsc:= store[tlsc+1];
                   fill_future_list(flib+store[tlsc],rlsc+1);
                   fill_result_list(12{RET},0);
                   bn:= bn - 1;
                   goto 105
                 end
          end
     else begin {end of switch declaration}
            sflag:= 0;
            repeat tlsc:= tlsc - 2;
```

```
                fill_result_list(1,88604672+store[tlsc])
                 {2T 'stacked RLSC' A}
              until store[tlsc-1] <> 160{switch comma};
              tlsc:= tlsc - 1; unload_t_list_element(nid);
              label_declaration;
              fill_result_list(0,85983232+48) {1T 16X1};
              tlsc:= tlsc - 1;
              fill_future_list(flib+store[tlsc],rlsc)
            end;
       eflag:= 0;
       if dl <> 105{end} then goto 1;
       tlsc:= tlsc - 1;
       if tlsc = tlib + 1 then goto 1052;
       repeat read_next_symbol
       until (dl = 91{;}) or (dl = 84{else}) or (dl = 105{end});
       jflag:= 0; pflag:= 0; fflag:= 0; nflag:= 0;
       goto 2;

  106: {own}                                                   {KH}
       new_block_by_declaration;
       read_next_symbol;
       if dl = 109{real} then ibd:= 0 else ibd:= 1;
       read_until_next_delimiter;
       if nflag = 0 then goto 1102;
       goto 1082;

  107: {Boolean}                                               {KZ}
       goto 108{integer};

  108:  {integer}                                              {KZ}
        ibd:= 1;
        new_block_by_declaration;
        read_until_next_delimiter;
  1081: if nflag = 0
        then begin if dl = 110{array} then goto 1101;
                goto 112{procedure}
              end;
        {scalar:}
        if bn <> 0 then goto 1083;
  1082: {static addressing}
        id:= gvc;
        if ibd = 1
        then begin id:= id + d19; gvc:= gvc + 1 end
        else gvc:= gvc + 2;
        fill_name_list;
```

```
      if dl = 87{,}
      then begin read_until_next_delimiter;
             goto 1082
           end;
      goto 1;
1083: {dynamic addressing}
      id:= pnlv + d15;
      if ibd = 1
      then begin id:= id + d19;
             pnlv:= pnlv + 32; lvc:= lvc + 1
           end
      else begin pnlv:= pnlv + 2 * 32; lvc:= lvc + 2 end;
      fill_name_list;
      if dl = 87{,}
      then begin read_until_next_delimiter;
             goto 1083
           end;
      read_until_next_delimiter;
      if (dl <= 106{own}) or (dl > 109{real})
      then begin reservation_of_local_variables;
             goto 2
           end;
      if dl = 109{real} then ibd:= 0 else ibd:= 1;
      read_until_next_delimiter;
      if nflag = 1 then goto 1083 {more scalars};
      reservation_of_local_variables;
      if dl = 110{array} then goto 1101;
      goto 3;

109: {real}                                                    {KE}
     ibd:= 0;
     new_block_by_declaration;
     read_until_next_delimiter;
     if nflag = 1 then goto 1081;
     goto 2;

110:  {array}                                                  {KF}
      ibd:= 0;
      new_block_by_declaration;
1101: if bn <> 0 then goto 1103;
1102: {static bounds, constants only:}
      id:= 3 * d24;
      if ibd <> 0 then id:= id + d19;
      repeat arra:= nlsc; arrb:= tlsc;
        repeat {read identifier list:}
```

```
      read_until_next_delimiter; fill_name_list
    until dl = 100{[};
    arrc:= 0;
    fill_t_list(2-ibd); {delta[0]}
    repeat {read bound-pair list:}
      {lower bound:}
      read_until_next_delimiter;
      if dl <> 90 {:}
      then if dl = 64{+}
           then begin read_until_next_delimiter;
                  arrd:= inw
                end
           else begin read_until_next_delimiter;
                  arrd:= - inw
                end
      else arrd:= inw;
      arrc:= arrc - (arrd * store[tlsc-1]) mod d26;
      {upper bound:}
      read_until_next_delimiter;
      if nflag = 0
      then if dl = 65{-}
           then begin read_until_next_delimiter;
                  arrd:= - inw - arrd
                end
           else begin read_until_next_delimiter;
                  arrd:= inw - arrd
                end
      else arrd:= inw - arrd;
      if dl = 101{[}
      then fill_t_list(- ((arrd + 1) * store[tlsc-1]) mod d26)
      else fill_t_list(((arrd + 1) * store[tlsc-1]) mod d26)
    until dl = 101{]};
    arrd:= nlsc;
    repeat {construction of storage function in constant list:}
      store[nlib+arrd-1]:= store[nlib+arrd-1] + klsc;
      fill_constant_list(gvc); fill_constant_list(gvc+arrc);
      tlsc:= arrb;
      repeat fill_constant_list(store[tlsc]);
        tlsc:= tlsc + 1
      until store[tlsc-1] <= 0;
      gvc:= gvc - store[tlsc-1]; tlsc:= arrb;
      if store[nlib+arrd-2] mod d3 = 0
      then arrd:= arrd - 2 else arrd:= arrd - 3
    until arrd = arra;
    read_until_next_delimiter
```

```
      until dl <> 87{,};
      goto 91{;};
1103: {dynamic bounds,arithmetic expressions:}
      ic:= 0; aic:= 0; id:= 0;
      repeat aic:= aic + 1;
        read_until_next_delimiter;
        fill_name_list
      until dl <> 87{,};
      eflag:= 1; oflag:= 1;
      goto 8501;

111: {switch}                                                     {HE}
     reservation_of_arrays;
     sflag:= 1;
     new_block_by_declaration;
     goto 1;

112:  {procedure}                                                 {HY}
      reservation_of_arrays;
      new_block_by_declaration;
      fill_result_list(2,88080384+flsc) {2T 'flsc'};
      fill_t_list(flsc); flsc:= flsc + 1;
      read_until_next_delimiter; look_for_name;
      label_declaration; intro_new_block;
      new_block_by_declaration1;
      if dl = 91{;} then goto 1;
      {formal parameter list:}
      repeat read_until_next_delimiter; id:= pnlv + d15 + d16;
        fill_name_list; pnlv:= pnlv + 2 * d5 {reservation PARD}
      until dl <> 87;
      read_until_next_delimiter; {for ; after )}
1121: read_until_next_delimiter;
      if nflag = 1 then goto 2;
      if dl = 104{begin} then goto 3;
      if dl <> 115{value} then goto 1123 {specification part};
      {value part:}
      spe:= d26; {value flag}
1122: repeat read_until_next_delimiter; look_for_name;
        store[nlib+nid]:= store[nlib+nid] + spe
      until dl <> 87;
      goto 1121;
1123: {specification part:}
      if (dl = 113{string}) or (dl = 110{array})
      then begin spe:= 0; goto 1122 end;
      if (dl = 114{label}) or (dl = 111{switch})
```

```
      then begin spe:= d17; goto 1122 end;
      if dl = 112{procedure}
      then begin spe:= d18; goto 1122 end;
      if dl = 109{real}
      then spe:= 0 else spe:= d19;
      if (dl <= 106) or (dl > 109) then goto 3; {if,for,goto}
      read_until_next_delimiter; {for delimiter following real/integer/boolean}
      if dl = 112{procedure}
      then begin spe:= d18; goto 1122 end;
      if dl = 110{array} then goto 1122;
1124: look_for_name; store[nlib+nid]:= store[nlib+nid] + spe;
      if store[nlib+nid] >= d26
      then begin id:= store[nlib+nid] - d26;
             id:= (id div d17) * d17 + id mod d16;
             store[nlib+nid]:= id;
             address_to_register; {generates 2S 'PARD position' A}
             if spe = 0
             then fill_result_list(14{TRAD},0)
             else fill_result_list(16{TIAD},0);
             address_to_register; {generates 2S 'PARD position' A}
             fill_result_list(35{TFR},0);
             fill_result_list(85{ST},0)
           end;
      if dl = 87{,}
      then begin read_until_next_delimiter;
             goto 1124
           end;
      goto 1121;

1052:
end {main_scan};


procedure program_loader;                                           {RZ}
var i,j,ll,list_address,id,mcp_count,crfa: integer;
    heptade_count,parity_word,read_location,stock: integer;
    from_store: 0..1;
    use: boolean;

  function logical_sum(n,m: integer): integer;
  {emulation of a machine instruction}
  var i,w: integer;
  begin w:= 0;
    for i:= 0 to 26 do
    begin w:= w div 2;
```

```
      if n mod 2 = m mod 2 then w:= w + d26;
      n:= n div 2; m := m div 2
    end;
    logical_sum:= w
  end {logical_sum};

  procedure complete_bitstock;                                    {RW}
  var i,w: integer;
  begin while bitcount > 0 {i.e., at most 20 bits in stock} do
    begin heptade_count:= heptade_count + 1;
      case from_store of
      0: {bit string read from store:}
         begin if heptade_count > 0
           then begin bitcount:= bitcount + 1;
                  heptade_count:= - 3;
                  read_location:= read_location - 1;
                  stock:= store[read_location];
                  w:= stock div d21;
                  stock:= (stock mod d21) * 64
                end
           else begin w:= stock div d20;
                  stock:= (stock mod d20) * 128
                end
         end;
      1: {bit string read from tape:}
         begin read(lib_tape,w);
           if heptade_count > 0
           then begin {test parity of the previous 4 heptades}
                  bitcount:= bitcount + 1;
                  parity_word:=
                    logical_sum(parity_word,parity_word div d4)
                    mod d4;
                  if parity_word in [0,3,5,6,9,10,12,15]
                  then stop(105);
                  heptade_count:= -3; parity_word:= w;
                  w:= w div 2
                end
           else parity_word:= logical_sum(parity_word,w)
         end
      end {case};
      for i:= 1 to bitcount - 1 do w:= 2 * w;
      bitstock:= bitstock + w; bitcount:= bitcount - 7
    end {while}
  end {complete_bitstock};
```

```
function read_bit_string(n: integer): integer;                        {RW}
var i,w: integer;
begin w:= 0;
  for i:= 1 to n do
  begin w:= 2 * w + bitstock div d26;
    bitstock:= (bitstock mod d26) * 2
  end;
  read_bit_string:= w; bitcount:= bitcount + n;
  complete_bitstock
end {read_bit_string};

procedure prepare_read_bit_string1;
var i: integer;
begin for i:= 1 to 27 - bitcount do bitstock:= 2 * bitstock;
  bitcount:= 21 - bitcount; heptade_count:= 0;
  from_store:= 0; complete_bitstock
end {prepare_read_bit_string1};

procedure prepare_read_bit_string2;
begin bitstock:= 0; bitcount:= 21; heptade_count:= 0;
  from_store:= 0; complete_bitstock;
  repeat until read_bit_string(1) = 1
end {prepare_read_bit_string2};

procedure prepare_read_bit_string3;
var w: integer;
begin from_store:= 1; bitstock:= 0; bitcount:= 21;
  repeat read(lib_tape,w) until w <> 0;
  if w <> 30 {D} then stop(106);
  heptade_count:= 0; parity_word:= 1;
  complete_bitstock;
  repeat until read_bit_string(1) = 1
end {prepare_read_bit_string3};

function address_decoding: integer;                                   {RY}
var w,a,n: integer;
begin w:= bitstock;
  if w < d26 {code starts with 0}
  then begin {0}      n:= 1; a:= 0; w:= 2 * w end
  else begin {1xxxxx} n:= 6; a:= (w div d21) mod d5;
         w:= (w mod d21) * d6
       end;
  if w < d25 {00}
  then begin {00} n:= n + 2; a:= 32 * a + 0; w:= w * 4 end else
  if w < d26 {01}
```

```
  then begin {01xx} n:= n + 4; a:= 32 * a + w div d23;
         if a mod d5 < 6
         then {010x} a:= a - 3 else {011x} a:= a - 2;
         w:= (w mod d23) * d4
       end
  else begin {1xxxxx} n:= n + 6;
         a:= a * 32 + (w div d21) mod d5;
         w:= (w mod d21) * d6
       end;
  if w < d25 {00}
  then begin {00} n:= n + 2; a:= 32 * a + 1 end else
  if w < d26 {01}
  then begin {01x} n:= n + 3; a := 32 * a + w div d24 end
  else begin {1xxxxx} n:= n + 6;
         a:= 32 * a + (w div d21) mod d5
       end;
  w:= read_bit_string(n); address_decoding:= a
end {address_decoding};

function read_mask: integer;                                         {RN}
var c: 0 .. 19;
begin
  if bitstock < d26 {code starts with 0}
  then {0x} c:= read_bit_string(2) else
  if bitstock < d26 + d25 {01}
  then {10x} c:= read_bit_string(3) - 2
  else {11xxxx} c:= read_bit_string(6) - 44;
  case c of
     0: read_mask:=   656; {0,   2S 0    A  }
     1: read_mask:= 14480; {3,   2B 0    A  }
     2: read_mask:= 10880; {2,   2T 0 X0    }
     3: read_mask:=  2192; {0,   2B 0    A  }
     4: read_mask:=   144; {0,   2A 0    A  }
     5: read_mask:= 10368; {2,   2B 0 X0    }
     6: read_mask:=  6800; {1,   2T 0    A  }
     7: read_mask:=     0; {0,   0A 0 X0    }
     8: read_mask:= 12304; {3,   0A 0    A  }
     9: read_mask:= 10883; {2, N 2T 0 X0    }
    10: read_mask:=  6288; {1,   2B 0    A  }
    11: read_mask:=  4128; {1,   0A 0 X0 B  }
    12: read_mask:=  8832; {2,   2S 0 X0    }
    13: read_mask:=   146; {0, Y 2A 0    A  }
    14: read_mask:=   256; {0,   4A 0 X0    }
    15: read_mask:=   134; {0, Y 2A 0 X0   P}
    16: read_mask:=   402; {0, Y 6A 0    A  }
```

```
      17: read_mask:=  4144; {1,   0A 0 X0 C  }
      18: read_mask:=    16; {0,   0A 0    A  }
      19: read_mask:= address_decoding
    end {case}
  end {read_mask};

  function read_binary_word: integer;                                    {RF}
  var w: integer; opc: 0 .. 3;
  begin if bitstock < d26 {code starts with 0}
    then begin {OPC >= 8}
           if bitstock < d25 {00}
           then if bitstock < d24 {000}
                then w:= 4 {code is 000x}
                else w:= 5 {code is 001xx}
           else if bitstock < d25 + d24 {010}
                then if bitstock < d25 + d23 {0100}
                     then w:= 6 {0100xx}
                     else w:= 7 {0101xxx}
                else w:= 10 {011xxxxxxx};
           w:= read_bit_string(w);
           if w <  2 {000x}    then {no change} else
           if w <  8 {001xx}   then w:= w -   2 else
           if w < 24 {010xx}   then w:= w -  10 else
           if w < 48 {0101xxx} then w:= w -  30
                else {011xxxxxxx}   w:= w - 366;
           read_binary_word:= opc_table[w]
         end {0}
    else begin w:= read_bit_string(1);
           w:= read_mask; opc:= w div d12;
           w:= (w mod d12) * d15 + address_decoding;
           case opc of
             0: ;
             1: w:= w + list_address;
             2: begin if w div d17 mod 2 = 1 {d17 = 1}
                  then w:= w - d17
                  else w:= w + d19;
                  w:= w  - w mod d15 + store[flib + w mod d15]
                end;
             3: if klib = crfb
                then w:= w - w mod d15 + store[mlib+w mod d15]
                else w:= w + klib
           end {case};
           read_binary_word:= w
         end {1}
  end {read_binary_word};
```

```
  procedure test_bit_stock;                                         {RH}
  begin if bitstock <> 63 * d21 then stop(107)
  end {test_bit_stock};

  procedure typ_address(a: integer);                                {RT}
  begin writeln(output);
    write(output,a div 1024:2,' ',(a mod 1024) div 32:2,' ',a mod 32:2)
  end {typ_address};

  procedure read_list;                                              {RL}
  var i,j,w: integer;
  begin for i:= ll - 1 downto 0 do
    begin w:= read_binary_word;
      if list_address + i <= flib + flsc
      then begin {shift FLI downwards}
             if flib <= read_location
             then stop(98);
             for j:= 0 to flsc - 1 do
             store[read_location+j]:= store[flib+j];
             flib:= read_location
           end;
      store[list_address+i]:= w
    end {for i};
    test_bit_stock;
  end {read_list};

  function read_crf_item: integer;                                  {RS}
  begin if crfa mod 2 = 0
    then read_crf_item:= store[crfa div 2] div d13
    else read_crf_item:= store[crfa div 2] mod d13;
    crfa:= crfa + 1
  end {read_crf_item};

begin {of program loader}
  rlib:= (klie - rlsc - klsc) div 32 * 32;
{increment entries in future list:}
  for i:= 0 to flsc - 1 do store[flib+i]:= store[flib+i] + rlib;
{move KLI to final position:}
  for i:= klsc - 1 downto 0 do store[rlib+rlsc+i]:= store[klib+i];
  klib:= rlib + rlsc;
{prepare mcp-need analysis:}
  mcpe:= rlib; mcp_count:= 0;
  for i:= 0 to 127 do store[mlib+i]:= 0;
{determine primary need of MCP's from name list:}
```

```
  i:= nlsc0;
  while i > nlscop do
  begin id:= store[nlib+i-1];
    if store[nlib+i-2] mod d3 = 0
    then {at most 4 letter/digit identifier} i:= i - 2
    else {at least 5 letters or digits} i:= i - 3;
    if (id div d15) mod 2 = 0
    then begin {MCP is used} mcp_count:= mcp_count + 1;
           store[mlib+(store[flib+id mod d15]-rlib) mod d15]:=
             - (flib + id mod d15)
         end
  end;
{determine secondary need using the cross-reference list:}
  crfa:= 2 * crfb;
  ll:= read_crf_item {for MCP length};
  while ll <> 7680 {end marker} do
  begin i:= read_crf_item {for MCP number};
    use:= (store[mlib+i] <> 0);
    j:= read_crf_item {for number of MCP needing the current one};
    while j <> 7680 {end marker} do
    begin use:= use or (store[mlib+j] <> 0); j:= read_crf_item end;
    if use
    then begin mcpe:= mcpe - ll;
           if mcpe <= mcpb then stop(25);
           if store[mlib+i] < 0
           then {primary need} store[-store[mlib+i]]:= mcpe
           else {only secondary need} mcp_count:= mcp_count + 1;
           store[mlib+i]:= mcpe
         end;
    ll:= read_crf_item
  end;
{load result list RLI:}
  ll:= rlsc; read_location:= rnsb;
  prepare_read_bit_string1;
  list_address:= rlib; read_list;
  if store[rlib] <> opc_table[89{START}] then stop(101);
  typ_address(rlib);
{copy MLI:}
  for i:= 0 to 127 do store[crfb+i]:= store[mlib+i];
  klib:= crfb; flsc:= 0;
{load MCP's from store:}
  prepare_read_bit_string2;
  ll:= read_bit_string(13) {for length or end marker};
  while ll < 7680 do
  begin i:= read_bit_string(13) {for MCP number};
```

```
    list_address:= store[crfb+i];
    if list_address <> 0
    then begin read_list; test_bit_stock;
           mcp_count:= mcp_count - 1;
           store[crfb+i]:= 0
         end
    else repeat read_location:= read_location - 1
         until store[read_location] = 63 * d21;
    prepare_read_bit_string2; ll:= read_bit_string(13)
  end;
{load MCP's from tape:}
  reset(lib_tape);
  while mcp_count <> 0 do
  begin writeln(output);
    writeln(output,'load (next) library tape into the tape reader');
    prepare_read_bit_string3;
    ll:= read_bit_string(13) {for length or end marker};
    while ll < 7680 do
    begin i:= read_bit_string(13) {for MCP number};
      list_address:= store[crfb+i];
      if list_address <> 0
      then begin read_list; test_bit_stock;
             mcp_count:= mcp_count - 1;
             store[crfb+i]:= 0
           end
      else repeat repeat read(lib_tape,ll) until ll = 0;
             read(lib_tape,ll)
           until ll = 0;
      prepare_read_bit_string3; ll:= read_bit_string(13)
    end
  end;
{program loading completed:}
  typ_address(mcpe)
end {program_loader};


{main program}

begin
{initialization of word_del_table}                                  {HT}
  word_del_table[10]:= 15086; word_del_table[11]:=    43;
  word_del_table[12]:=     1; word_del_table[13]:=    86;
  word_del_table[14]:= 13353; word_del_table[15]:= 10517;
  word_del_table[16]:=    81; word_del_table[17]:= 10624;
  word_del_table[18]:=    44; word_del_table[19]:=     0;
```

```
  word_del_table[20]:=     0; word_del_table[21]:= 10866;
  word_del_table[22]:=     0; word_del_table[23]:=     0;
  word_del_table[24]:=   106; word_del_table[25]:=   112;
  word_del_table[26]:=     0; word_del_table[27]:= 14957;
  word_del_table[28]:=     2; word_del_table[29]:=     2;
  word_del_table[30]:=    95; word_del_table[31]:=   115;
  word_del_table[32]:= 14304; word_del_table[33]:=     0;
  word_del_table[34]:=     0; word_del_table[35]:=     0;
  word_del_table[36]:=     0; word_del_table[37]:=     0;
  word_del_table[38]:=   107;

{initialization of flex_table}                                           {LK}
  flex_table[  0]:=    -2; flex_table[  1]:= 19969; flex_table[  2]:= 16898;
  flex_table[  3]:=    -0; flex_table[  4]:= 18436; flex_table[  5]:=    -0;
  flex_table[  6]:=    -0; flex_table[  7]:= 25863; flex_table[  8]:= 25096;
  flex_table[  9]:=    -0; flex_table[ 10]:=    -0; flex_table[ 11]:=    -1;
  flex_table[ 12]:=    -0; flex_table[ 13]:=    -1; flex_table[ 14]:= 41635;
  flex_table[ 15]:=    -0; flex_table[ 16]:= 31611; flex_table[ 17]:=    -0;
  flex_table[ 18]:=    -0; flex_table[ 19]:= 17155; flex_table[ 20]:=    -0;
  flex_table[ 21]:= 23301; flex_table[ 22]:= 25606; flex_table[ 23]:=    -0;
  flex_table[ 24]:=    -0; flex_table[ 25]:= 25353; flex_table[ 26]:= 30583;
  flex_table[ 27]:=    -0; flex_table[ 28]:=    -1; flex_table[ 29]:=    -0;
  flex_table[ 30]:=    -0; flex_table[ 31]:=    -1; flex_table[ 32]:= 19712;
  flex_table[ 33]:=    -0; flex_table[ 34]:=    -0; flex_table[ 35]:= 14365;
  flex_table[ 36]:=    -0; flex_table[ 37]:= 14879; flex_table[ 38]:= 15136;
  flex_table[ 39]:=    -0; flex_table[ 40]:=    -0; flex_table[ 41]:= 15907;
  flex_table[ 42]:=    -1; flex_table[ 43]:=    -0; flex_table[ 44]:=    -1;
  flex_table[ 45]:=    -0; flex_table[ 46]:=    -0; flex_table[ 47]:=    -1;
  flex_table[ 48]:=    -0; flex_table[ 49]:= 17994; flex_table[ 50]:= 14108;
  flex_table[ 51]:=    -0; flex_table[ 52]:= 14622; flex_table[ 53]:=    -0;
  flex_table[ 54]:=    -0; flex_table[ 55]:= 15393; flex_table[ 56]:= 15650;
  flex_table[ 57]:=    -0; flex_table[ 58]:=    -0; flex_table[ 59]:= 30809;
  flex_table[ 60]:=    -0; flex_table[ 61]:=    -1; flex_table[ 62]:= 30326;
  flex_table[ 63]:=    -0; flex_table[ 64]:= 19521; flex_table[ 65]:=    -0;
  flex_table[ 66]:=    -0; flex_table[ 67]:= 12309; flex_table[ 68]:=    -0;
  flex_table[ 69]:= 12823; flex_table[ 70]:= 13080; flex_table[ 71]:=    -0;
  flex_table[ 72]:=    -0; flex_table[ 73]:= 13851; flex_table[ 74]:=    -1;
  flex_table[ 75]:=    -0; flex_table[ 76]:=    -1; flex_table[ 77]:=    -0;
  flex_table[ 78]:=    -0; flex_table[ 79]:=    -1; flex_table[ 80]:=    -0;
  flex_table[ 81]:= 11795; flex_table[ 82]:= 12052; flex_table[ 83]:=    -0;
  flex_table[ 84]:= 12566; flex_table[ 85]:=    -0; flex_table[ 86]:=    -0;
  flex_table[ 87]:= 13337; flex_table[ 88]:= 13594; flex_table[ 89]:=    -0;
  flex_table[ 90]:=    -0; flex_table[ 91]:= 31319; flex_table[ 92]:=    -0;
  flex_table[ 93]:=    -1; flex_table[ 94]:=    -1; flex_table[ 95]:=    -0;
  flex_table[ 96]:=    -0; flex_table[ 97]:=  9482; flex_table[ 98]:=  9739;
```

```
    flex_table[ 99]:=    -0; flex_table[100]:= 10253; flex_table[101]:=    -0;
    flex_table[102]:=    -0; flex_table[103]:= 11024; flex_table[104]:= 11281;
    flex_table[105]:=    -0; flex_table[106]:=    -0; flex_table[107]:= 31832;
    flex_table[108]:=    -0; flex_table[109]:=    -1; flex_table[110]:=    -1;
    flex_table[111]:=    -0; flex_table[112]:= 31040; flex_table[113]:=    -0;
    flex_table[114]:=    -0; flex_table[115]:=  9996; flex_table[116]:=    -0;
    flex_table[117]:= 10510; flex_table[118]:= 10767; flex_table[119]:=    -0;
    flex_table[120]:=    -0; flex_table[121]:= 11538; flex_table[122]:=    -2;
    flex_table[123]:=    -0; flex_table[124]:=    -2; flex_table[125]:=    -0;
    flex_table[126]:=    -0; flex_table[127]:=    -2;

  {preparation of prescan}                                          {LE}
    rns_state:= virginal; scan:= 1;
    read_until_next_delimiter;

    prescan;                                                        {HK}

    {writeln;
    for bn:= plib to plie do writeln(bn:5,store[bn]:10);
    writeln;}

  {preparation of main scan:}                                       {HL}
    rns_state:= virginal; scan:= - 1;
    iflag:= 0; mflag:= 0; vflag:= 0; bn:= 0; aflag:= 0; sflag:= 0;
    eflag:= 0; rlsc:= 0; flsc:= 0; klsc:= 0; vlam:= 0;
    flib:= rnsb + 1; klib:= flib + 16; nlib:= klib + 16;
    if nlib + nlsc0 >= plib then stop(25);
    nlsc:= nlsc0; tlsc:= tlib; gvc:= gvc0;
    fill_t_list(161);
  {prefill of name list:}
    store[nlib +  0]:= 27598040;
    store[nlib +  1]:=   265358;             {read}
    store[nlib +  2]:= 134217727 -        6;
    store[nlib +  3]:= 61580507;
    store[nlib +  4]:=   265359;             {print}
    store[nlib +  5]:= 134217727 -  53284863;
    store[nlib +  6]:=   265360;             {TAB}
    store[nlib +  7]:= 134217727 -  19668591;
    store[nlib +  8]:=   265361;             {NLCR}
    store[nlib +  9]:= 134217727 -        0;
    store[nlib + 10]:= 134217727 -  46937177;
    store[nlib + 11]:=   265363;             {SPACE}
    store[nlib + 12]:= 53230304;
    store[nlib + 13]:=   265364;             {stop}
    store[nlib + 14]:= 59085824;
```

```
store[nlib + 15]:=   265349;               {abs}
store[nlib + 16]:= 48768224;
store[nlib + 17]:=   265350;               {sign}
store[nlib + 18]:= 61715680;
store[nlib + 19]:=   265351;               {sqrt}
store[nlib + 20]:= 48838656;
store[nlib + 21]:=   265352;               {sin}
store[nlib + 22]:= 59512832;
store[nlib + 23]:=   265353;               {cos}
store[nlib + 24]:= 48922624;
store[nlib + 25]:=   265355;               {ln}
store[nlib + 26]:= 53517312;
store[nlib + 27]:=   265356;               {exp}
store[nlib + 28]:= 134217727 -       289;
store[nlib + 29]:= 29964985;
store[nlib + 30]:=   265357;               {entier}

store[nlib + 31]:= 134217727 -  29561343;
store[nlib + 32]:=   294912;               {SUM}
store[nlib + 33]:= 134217727 -  14789691;
store[nlib + 34]:= 134217727 -  15115337;
store[nlib + 35]:=   294913;               {PRINTTEXT}
store[nlib + 36]:= 134217727 -  27986615;
store[nlib + 37]:=   294914;               {EVEN}
store[nlib + 38]:= 134217727 -       325;
store[nlib + 39]:= 21928153;
store[nlib + 40]:=   294915;               {arctan}
store[nlib + 41]:= 134217727 -  15081135;
store[nlib + 42]:=   294917;               {FLOT}
store[nlib + 43]:= 134217727 -  14787759;
store[nlib + 44]:=   294918;               {FIXT}
store[nlib + 45]:= 134217727 -      3610;
store[nlib + 46]:= 134217727 -  38441163;
store[nlib + 47]:=   294936;               {ABSFIXT}

intro_new_block2;
bitcount:= 0; bitstock:= 0; rnsb:= bim;
fill_result_list(96{START},0);
pos:= 0;
main_scan;                                                      {EL}
fill_result_list(97{STOP},0);

{writeln; writeln('FLI:');
for bn:= 0 to flsc-1 do
writeln(bn:5,store[flib+bn]:10);}
```

```
    {writeln; writeln('KLI:');
    for bn:= 0 to klsc-1 do
    writeln(bn:5,store[klib+bn]:10,
           (store[klib+bn] mod 134217728) div 16777216 : 10,
                      (store[klib+bn] mod  16777216) div  2097152 : 2,
                      (store[klib+bn] mod   2097152) div   524288 : 3,
                      (store[klib+bn] mod    524288) div   131072 : 2,
                      (store[klib+bn] mod    131072) div    32768 : 2,
                      (store[klib+bn] mod     32768) div     1024 : 4,
                      (store[klib+bn] mod      1024) div       32 : 3,
                      (store[klib+bn] mod        32) div        1 : 3);}

  {preparation of program loader}
    opc_table[ 0]:=  33; opc_table[ 1]:=  34; opc_table[ 2]:=  16;
    opc_table[ 3]:=  56; opc_table[ 4]:=  58; opc_table[ 5]:=  85;
    opc_table[ 6]:=   9; opc_table[ 7]:=  14; opc_table[ 8]:=  18;
    opc_table[ 9]:=  30; opc_table[10]:=  13; opc_table[11]:=  17;
    opc_table[12]:=  19; opc_table[13]:=  20; opc_table[14]:=  31;
    opc_table[15]:=  35; opc_table[16]:=  39; opc_table[17]:=  61;
    opc_table[18]:=   8; opc_table[19]:=  10; opc_table[20]:=  11;
    opc_table[21]:=  12; opc_table[22]:=  15;
    for ii:= 23 to 31 do opc_table[ii]:= ii - 2;
    opc_table[32]:=  32; opc_table[33]:=  36; opc_table[34]:=  37;
    opc_table[35]:=  38;
    for ii:= 36 to 51 do opc_table[ii]:= ii + 4;
    opc_table[52]:=  57; opc_table[53]:=  59; opc_table[54]:=  60;
    for ii:= 55 to 102 do opc_table[ii]:= ii + 7;

    store[crfb+ 0]:=   30 * d13 +    0; store[crfb+ 1]:= 7680 * d13 +   20;
    store[crfb+ 2]:=    1 * d13 + 7680; store[crfb+ 3]:=   12 * d13 +    2;
    store[crfb+ 4]:= 7680 * d13 +   63; store[crfb+ 5]:=    3 * d13 + 7680;
    store[crfb+ 6]:=   15 * d13 +    4; store[crfb+ 7]:=    3 * d13 + 7680;
    store[crfb+ 8]:=  100 * d13 +    5; store[crfb+ 9]:= 7680 * d13 +  134;
    store[crfb+10]:=    6 * d13 +   24; store[crfb+11]:= 7680 * d13 +   21;
    store[crfb+12]:=   24 * d13 + 7680; store[crfb+13]:= 7680 * d13 + 7680;

    store[mcpb]:= 63 * d21; store[mcpb+1]:= 63 * d21;

    program_loader;

    writeln(output); writeln(output); writeln(output);
    for ii:= mcpe to rlib + rlsc + klsc - 1 do
    writeln(output,ii:5,store[ii]:9)
```

```
end.
```

Chapter 10

The X1–code version of the compiler

The following text is as it occurred in manuscript. When punched for producing a tape to be assembled by the X1 assembler, all commentary and all lay–out symbols had to be left out. So, with some exceptions, only columns 12 to 28 are relevant as X1–code.

```
Voorponsingen                                                          VPO


     DP ZZ 7298 X 0       7-04-02          vertaler, 1ste en 2de doorgang
     DP ZE 6784 X 0       6-20-00          werkruimte 1ste en 2de doorgang
     DP RZ  0   X 7       0-07-00          vertaler, 3de doorgang
     DP SF 11   X 3       0-03-11          OPC-tabel



     DP ZF  1 Z Z 0       7-04-03          FRL
     DP ZH  5 Z F 2       7-06-08          GAI
     DP ZK 13 Z H 0       7-06-21          constanten
     DP ZL 31 Z K 0       7-07-20          PTM
     DP ZR  6 Z L 0       7-07-26          AVR/BPR
     DP ZS 13 Z R 0       7-08-07          POP
     DP ZT 15 Z S 3       7-11-22          FTL
     DP ZW  7 Z T 0       7-11-29          FTD
     DP ZU  4 Z W 0       7-12-01          LTF
     DP ZY  7 Z U 0       7-12-08          RNS
     DP ZN 19 Z Y 1       7-13-27          THENELSE



     DP EZ 15 Z N 0       7-14-10          DDEL :=
     DP EE 14 E Z 2       7-16-24          DDEL [
     DP EF 27 E E 0       7-17-19          DDEL ]
     DP EH  7 E F 2       7-19-26          DDEL
     DP EK 15 E H 1       7-21-09          DDEL ,
     DP EL 17 E K 4       7-25-26          BASIC CYCLE
     DP ER  8 E L 0       7-26-02          DOT
     DP ES 20 E R 0       7-26-22          DDEL + -
     DP ET 14 E S 0       7-27-04          DDEL * / div
     DP EW  4 E T 0       7-27-08          DDEL (
     DP EU 13 E W 1       7-28-21          DDEL )
     DP EY 10 E U 0       7-28-31          DDEL if
     DP EN 12 E Y 0       7-29-11          DDEL then
```

VP1

```
DP FZ 20 E N 0        7-29-31        DDEL else
DP FE  4 F Z 1        7-31-03        DDEL for
DP FF 18 F E 0        7-31-21        DDEL while
DP FH  5 F F 0        7-31-26        DDEL step
DP FK  5 F H 0        7-31-31        DDEL until
DP FL  3 F K 0        8-00-02        DDEL do
DP FR  4 F L 1        8-01-06        ETT
DP FS  8 F R 0        8-01-14        DDEL ;  DDEL end
DP FT 15 F S 2        8-03-29        RND
DP FW 10 F T 5        8-09-07        LFC
DP FU 30 F W 1        8-11-05        FFL
DP FY 24 F U 0        8-11-29        LDEC
DP FN  0 F Y 2        8-13-29        DDEL :


DP HZ  9 F N 0        8-14-06        LFN
DP HE 11 H Z 1        8-15-17        DDEL switch
DP HF  5 H E 0        8-15-22        FPL
DP HH  3 H F 1        8-16-25        APL
DP HK  7 H H 0        8-17-00        PSP
DP HL  3 H K 4        8-21-03        EPS
DP HR 19 H L 1        8-22-22        FOB 6
DP HS  6 H R 1        8-23-28        OCT
DP HT 25 H S 0        8-24-21        NSS
DP HW 20 H T 4        8-29-09        INB
DP HU 14 H W 1        8-30-23        NBD
DP HY  8 H U 1        8-31-31        DDEL procedure
DP HN 11 H Y 3        9-03-10        FNL
```

```
                                                      VP2

DP KZ 16 H N 0        9-04-26         DDEL integer
DP KE 25 K Z 1        9-05-19         DDEL real
DP KF  7 K E 0        9-05-26         DDEL array
DP KH  3 K F 3        9-08-29         DDEL own
DP KK 10 K H 0        9-09-07         DDEL < <= = >= > <>
DP KL  4 K K 0        9-09-11         DDEL not and or implies eqv
DP KR  5 K L 0        9-09-16         DDEL goto
DP KS  2 K R 0        9-09-18         DDEL (*
DP KT 30 K S 0        9-10-16         DDEL **
DP KW  2 K T 0        9-10-18         END
DP KU 10 K W 3        9-13-28         FKL
DP KY 22 K U 0        9-14-18         RLV
DP KN 13 K Y 0        9-14-31         RLA

DP LZ  3 K N 2        9-17-02         DDEL begin
DP LE  5 L Z 0        9-17-07         SPS
DP LF  9 L E 0        9-17-16         TFO
DP LH 22 L F 0        9-18-06         PST
DP LK 19 L H 0        9-18-25         RFS
DP LL  0 L K 5        9-23-25         BSM
DP LR 14 L L 1        9-25-07         CWD
DP LS 22 L R 1        9-26-29         ADC
DP LT  9-Z Z 0        7-03-25         werkruimte fano-codering
DP LW  3 L S 1        9-28-00         eerste vrije plaats
```

VP3

```
DP RE 26 R Z11        0-18-26          werkruimte derde doorgang
DP RF  6 R Z 2        0-09-06          RBW
DP RH 27 R F 1        0-11-01          TBV
DP RK  5 R H 0        0-11-06          constanten deel 2
DP RL  6 R K 0        0-11-12          LIL
DP RR 26 R L 0        0-12-06          LLN
DP RS  4 R R 0        0-12-10          HSC
DP RT 11 R S 0        0-12-21          TYP
DP RW 13 R T 0        0-13-02          RBS
DP RU 18 R W 2        0-15-20          MCPL
DP RY 22 R U 0        0-16-10          ADD
DP RN  8 R Y 1        0-17-18          ML


DP SZ 21 R N 0        0-18-07          MT
DP SE 15 R E 0        0-19-09          (ZE) werkruimte derde doorgang
```

```
    SAT    start ALGOL translation                                      ZZ0

    aanroep                          autostart 0


         DA   0 Z Z 0       DI
=>>  0   2T   0 L E 0 A     =>        start prescan
         DC D0
```

```
    FRL     fill result list                                          ZF0

    aanroep                        6T 0 Z F0 0  =)  FRL


         DA   0 Z F 0       DI
  =)  0   2B   1       A             RLSC:= RLSC + 1
      1   4B  24 Z E 0
      2 U 1A   7       A P
      5 N 2T  14 Z F 0 A     ->      als OPCnr < 8
      4 U 1A  61       A P
      5 Y 2T   9 Z F 0 A     ->      als OPCnr > 61
      6   4P       AB                voor 8 <= OPCnr <= 61:
      7   2S   8-L R 0 B             zoek codewoord in tabel en
      8   2T   0 L L 0 A     =>      BSM met OPCcode
 5 => 9 U 1A  85       A Z           voor 61 < OPCnr :
     10 Y 2S  5127     A             bouw zelf het codewoord op
     11 N 4P       AS
     12 N 0S  10599    A
     13   2T   0 L L 0 A     =>      BSM met OPCcode
 3 => 14   6A   8 L T 0              berg OPCnr (0,1,2,3)
     15   2A   8   X 0               transport link
     16   6A   7 L T 0
     17   6T   0 L S 0 0     =)      ADC voor adresgedeelte
     18   3LS  32767   A             isoleer functiegedeelte
     19   0S   8 L T 0               en voeg OPCnr als adres toe
     20   2B  19       A
23 -> 21 U 0LS 30 Z F 0 B Z
     22 N 1B   1       A Z           cyclus test op masker
     23 N 2T  21 Z F 0 A     ->
     24   4P       BB    P           masker gevonden?
     25 Y 2S  17 Z F 1 B              zo ja, pak maskercode uit tabel
     26 N 0P  12   SS                zo neen,
     27 N 6T   0 L S 0 0     =)      ADC voor functiegedeelte
     28 N 2S  7295     A             en pak speciale maskercode
     29   6T   0 L L 0 0     =)      BSM met maskercode
     30   2T   7 L T 0   E   =>      klaar
     31   0A   0       A             masker nr  1
          DC D0
```

```
                                                                   ZF1

      DA   0 Z F 1       DI
 0    0A   1   X 0 C               masker nr  2
 1 Y  6A   0       A                          3
 2 Y  2A   0   X 0   P                        4
 3    4A   0   X 0                            5
 4 Y  2A   0       A                          6
 5    2S   2   X 0                            7
 6    0A   1   X 0 B                          8
 7    2B   1       A                          9
 8 N  2T   2   X 0                           10
 9    0A   3       A                         11
10    0A   0   X 0                           12
11    2T   1       A                         13
12    2B   2   X 0                           14
13    2A   0       A                         15
14    2B   0       A                         16
15    2T   2   X 0                           17
16    2B   3       A                         18
17    2S   0       A     DN                  19
18         +7294                   maskercode nr  1
19         +7293                                  2
20         +7292                                  3
21         +7291                                  4
22         +7290                                  5
23         +7289                                  6
24         +7288                                  7
25         +7287                                  8
26         +7286                                  9
27         +7285                                 10
28         +7284                                 11
29         +7283                                 12
30         +7282                                 13
31         +7281                                 14
      DC D0
```

ZF2

```
   DA   0 Z F 2       DN
0      +7280                 maskercode nr 15
1      +4109                               16
2      +4108                               17
3      +3077                               18
4      +3076                               19
   DC D0
```

```
GAI     generate address ID in next accumulator                     ZH0

aanroep                          6T 0 Z H0 1  =)  GAI



         DA   0 Z H 0       DI
  =)  0   6T   0 Z R 0 0     =)        AVR
      1   2S   8 Z E 0                 pak ID
      2 U 2LS  3 Z K 0   Z             non-formele?
      3 N 2A  18       A               OPC 18: TFA
      4 N 2T  10 Z H 0 A     ->
      5   2A  14       A
      6 U 2LS  0 Z K 0   Z             statisch?
      7 Y 0A   1       A
      8 U 2LS  4 Z K 0   Z             real?
      9 N 0A   2       A               OPC 14,15,16 of 17
 4 -> 10   2S   0       A
     11   6T   0 Z F 0 0     =)        FRL
     12   2T   9   X 0   E   =>        klaar
          DC D0
```

```
     DA   0 Z K 0       DN
 0       +32768         DI        = d15
 1   2B   0       A
 2   2S   0       A     DN
 3       +65536                   = d16
 4       +524288        DI        = d19
 5   2B   0   X 0
 6   2A   0       A
 7   2T   0       A
 8   2T   0   X 0       DN
 9       +17825792                = d24,d20
10       +18350080      DI        = d24,d20,d19
11   0D  32767 X 0      DN        = d25,d24,d14/d0
12       +131072        DI        = d17
13   0X   0   X 0                 = d25
14   0D   0   X 0       DN        = d25,d24
15       +1048576       DI        = d20
16 N 2T   0   X 0
17   2S   0   X 0
18   4A   0   X 0                 = d23
19   6D   0   X 0                 = d25/d22
20   1T  16   X 1
21   0S   0   X 0       DN        = d24
22       +65535         DI        = d15/d0
23 U 0A   0   X 0   Z             = d18,d15
24 U 0A   0   X 0   P             = d17,d15
25   4A  17   X 1
26   4A  18   X 1
27 N 0A   0   X 0                 = d16,d15
28   0B   0   X 0                 = d26
29   0A   0   X 0   Z             = d18
30 U 0Y   0   X 0                 = d26,d25,d15
     DC D0
```

```
PTM     production transmark                                          ZL0

aanroep                         6T 0 Z L0 0  =)  PTM



        DA   0 Z L 0       DI
=)  0   2A  19 Z E 0                FFLA
    1   0P   1   AA
    2   1A  18 Z E 0                2*FFLA - EFLA
    3   0A   9       A
    4   2S   0       A              met OPC 8,9,10 of 11
    5   2T   0 Z F 0 A     =>       door naar FRL
        DC D0
```

```
BPR     begin procedure in register =                             ZR0
AVR     address variable in register

aanroep                         6T 0 Z R0 0  =)  BPR of AVR



     DA   0 Z R 0       DI
=) 0   2A   8 Z E 0                pak ID
   1   4P       AS
   2   2LS 32767    A
   3 U 2LA  0 Z K 0   Z            statisch?
   4 N 0S   2 Z K 0                zo neen,
   5 N 2A   0       A              met 2S dynamisch adres A
   6 N 2T   0 Z F 0 A     ->       door naar FRL
   7   0P   3   AA
   8   2LA  3       A
   9 U 0LA  2       A Z            verwijzing naar FLI?
  10 Y 0S   5 Z K 0                met 2B FLIadres
  11 N 0S   1 Z K 0                of  2B statisch adres A
  12   2T   0 Z F 0 A     =>       door naar FRL
       DC D0
```

```
        POP     production object program                                    ZS0

        aanroepen                         6T  0 Z S0 1  =)  OH:= 1; POP
                                          6T  1 Z S0 1  =)  OH:= [A]; POP
                                          6T  2 Z S0 1  =)  POP



             DA   0 Z S 0       DI
     =)  0   2A   1       A
     =)  1   6A   5 Z E 0                 OH:= 1
     =)  2   2S   29 Z E 0   Z            NFLA = 0?
         3 N 2T   16 Z S 1 A     ->       zo neen, dan naam of konstante
         4   2A   13 Z E 0   Z            AFLA = 0 ?
         5 N 2A   58      A               zo neen, dan
         6 N 6S   13 Z E 0                AFLA:= 0, en
18LH0    7 N 6T   0 Z F 0 0     =)        FRL met OPC 58: TAR
14   ->  8   2B   25 Z E 0
26ZS1    9   2S   32767 X 0 B             TLI[TLSC - 1]
        10   3P   8   SS
        11   2LS 15       A               isoleer OH uit TLI
        12 U 5S   5 Z E 0   P             en check deze tegen OH-heersend
        13 Y 2T   9   X 0   E    ->       als OH-heersend > OH-uit-TLI
        14   1B   1       A
        15   6B   25 Z E 0                TLSC:= TLSC - 1
        16   2A   0   X 0 B
        17   2LA 255      A               isoleer operator uit TLI
        18 U 1A   63      A P
        19 U 1A   80      A E             is het een adreshebbende?
27      20 Y 2T   25 Z S 0 A     ->
8,11ZS1 21   1A   5       A               adresloze operator: + t/m eqv
18ZS2-> 22   2S   0       A
7,14ZS3 23   6T   0 Z F 0 0     =)        FRL met OPC van operator
        24   2T   8 Z S 0 A     =>        volgend element uit TLI
  20 => 25 U 1A   132     A Z             operator = NEG?
        26 Y 2A   57      A               dan OPC van NEG
        27 Y 2T   22 Z S 0 A     ->       en verder als adresloze
        28 U 1A   127     A E             operator <> STORE?
        29 Y 2T   9 Z S 1 A     ->        dan speciale functie
        30 U 1A   129     A P             assignment to proc.identifier?
        31   6A   3 Z E 0
             DC D0
```

ZS1

```
             DA   0 Z S 1       DI
           0 Y 1B   1       A
           1 Y 6B  25 Z E 0                 zo ja, TLSC:= TLSC - 1
           2 Y 2S   0   X 0 B               pak BN uit TLI
           3 Y 0S   1 Z K 0
           4 Y 2A   0       A
           5 Y 6T   0 Z F 0 0     =)        FRL met  2B 'BN' A
           6   3A  43       A
           7   0A   3 Z E 0                 bouw OPCnr op
           8   2T  22 Z S 0 A     =>        en verder als adresloze
29ZS0=>    9 U 1A  141      A E
          10 N 1A  57       A               bouw OPCnr van
          11 N 2T  22 Z S 0 A     ->
          12 U 1A  151      A E             speciale functie op
          13 N 1A  40       A
          14 N 2T  22 Z S 0 A     ->        en verder als adresloze
          15   7Y   1   C 0                 stop: onbekende functie (OPC>111)
3ZS0 =>   16   2A   0       A
          17   6A  29 Z E 0                 NFLA:= 0
          18   6A  13 Z E 0                 AFLA:= 0
          19   2S  16 Z E 0   Z             PFLA = 0?
          20 Y 2T  27 Z S 1 A     ->        dan geen procedure statement
          21   6T   0 L H 0 3     =)        PST voor parameterloze procedure
          22   2S   6 Z K 0
          23   2A   0       A
          24   6T   0 Z F 0 0     =)        FRL met 2A  0  A
          25   6T   0 Z L 0 0     =)        PTM
          26   2T   8 Z S 0 A     =>        volgend element uit TLI
  20 =>   27   2A   2 Z E 0   Z             JFLA = 0?
          28 Y 2T  19 Z S 2 A     ->        als geen go to statement
          29   2A  19 Z E 0   Z             FFLA = 0? dwz. non formele?
          30 N 6T   0 Z R 0 0     =)        BPR voor formele label
          31 N 2A  35       A               OPC 35: TFR
             DC D0
```

ZS2

```
           DA   0 Z S 2       DI
         0 N 2T  22 Z S 0 A     ->       verder als adresloze operator
         1   2S   8 Z E 0                ID
         2   0P   8   SS
         3   2LS 31       A              isoleer BN uit ID
         4 U 1S   7 Z E 0   Z            blijven we in het block?
         5 N 0S   1 Z K 0                zo neen,
         6 N 6T   0 Z F 0 0     =)       FRL met  2B  'BN' A
         7 N 2S   0       A
         8 N 2A  28       A
         9 N 6T   0 Z F 0 0     =)       FRL met GTA
        10   6T   0 L F 0 0     =)       TFO
        11   4P       AS                 A en S bevatten nu ID
        12   2LS 32767    A              isoleer adres
        13   0P   3   AA
        14   2LA  3       A              isoleer opc
        15 U 0LA  2       A Z            referentie naar FLI?
        16 Y 0S   8 Z K 0                dan  2T  'adres'
        17 N 0S   7 Z K 0                anders  2T  'adres' A
        18   2T  23 Z S 0 A     =>       verder als adresloze operator
28ZS1=> 19   6T   0 Z R 0 0     =)       AVR
        20   2B  25 Z E 0
        21   2S  32767 X 0 B             TLI[TLSC - 1]
        22   4P       SA
        23   3P   8   SS
        24   2LS 15       A              isoleer OH uit TLI
        25 U 5S   5 Z E 0   P            OH-heersend > OH-uit-TLI?
  31 -> 26 Y 2A  315      A              5 * 63
        27 Y 2T   5 Z S 3 A     ->       produceer dan TAKE
        28   2LA 255      A              isoleer operator uit TLI
        29 U 1A  63       A P
        30 U 1A  67       A E            adresloze operator?
        31 Y 2T  26 Z S 2 A     ->       ga dan TAKE produceren
           DC D0
```

```
            DA   0 Z S 3       DI
         0  6A   0   X 1                anders een adreshebbende
         1  0P   2   AA                 operatie met ingebouwde TAKE
         2  0A   0   X 1                produceren uit 5 * operator
         3  1B   1       A
         4  6B  25 Z E 0                terwijl TLSC:= TLSC - 1
27ZS2->  5  2S  19 Z E 0   Z            FFLA = 0?
         6 N 1A  280     A              voor formele: - 5 * 64 + 40
         7 N 2T  22 Z S 0 A     ->      en verder als adresloze
         8  2S   8 Z E 0                voor non formele:
         9 U 2LS  0 Z K 0   Z           d15 van ID = 0?
        10 Y 0A   1       A             als statisch
        11 U 2LS  4 Z K 0   Z           d19 van ID = 0?
        12 N 0A   2       A             als integer type
        13  1A  284       A             - 5 * 64 + 36
        14  2T  22 Z S 0 A     =>       en verder als adresloze
            DC D0
```

```
FTL     fill TLI with (S)                                                ZT0

aanroep                           6T 0 Z T0 0  =)  FTL



         DA   0 Z T 0       DI
=)   0   2B  25 Z E 0                TLSC
     1 U 1B   1 R K 0   Z            = MCPB?
     2 Y 7Y   2   C 0                stop:   TLI vol
     3   6S   0   X 0 B              TLI[TLSC]:= (S)
     4   0B   1       A
     5   6B  25 Z E 0                TLSC:= TLSC + 1
     6   2T   8   X 0   E   =>       klaar
         DC D0
```

```
   FTD     fill TLI with delimiter                                  ZW0

   aanroep                          6T 0 Z W0 0  =)  FTD



        DA   0 Z W 0       DI
=)  0   2S   5 Z E 0               OH
    1   0P   8   SS
    2   0S   9 Z E 0               DL
    3   2T   0 Z T 0 A     =>      met 256*OH + DL door naar FTL
        DC D0
```

```
  LTF     inverse of FTL                                                  ZU0

  aanroep                           6T 0 Z U0 0  =)  LTF
                                    with (S) = destination of result



       DA   0 Z U 0       DI
=)  0  3B   1       A
    1  0B  25 Z E 0
    2  6B  25 Z E 0                 TLSC:= TLSC - 1
    3  2A   0   X 0 B               A:= TLI[TLSC]
    4  4P       SB
    5  6A   0   X 0 B
    6  2T   8   X 0   E   =>        klaar
       DC D0
```

```
     RNS     read next symbol into DL                                    ZY0

     aanroep                        6T 0 Z Y0 0  =)  RNS



          DA   0 Z Y 0       DI
   =)  0   0T  14 Z E 2       =>       strooisprong over toestand
  0 =>  1   2T  31 Z Y 0 A     =>       RNS maagdelijk, doe voorbereiding
  0 =>  2   2T   0 H T 0 A     =>       naar NSS, want prescan
NSS =>  3   6S   9 Z E 0                berg symbool in DL
        4   2B  12 Z E 2                oude schuifwijzer
        5   1B   7       A P
        6 N 2B  15       A
        7   6B  12 Z E 2                nieuwe schuifwijzer
        8   0P   0   SS  B              schuif
        9   2B  19 Z E 1                vulplaats
       10 Y 4S   0   X 0 B              als nog plaats in
       11 Y 2T   8   X 0   E  ->             oude woord dan klaar
       12   0B   1       A
       13   6B  19 Z E 1                nieuwe vulplaats
       14   6S   0   X 0 B              start nieuw magazijnwoord
       15   0B   8       A
       16   1B  21 Z E 0   P            vulplaats + 8 > PLIB?
       17 Y 7Y  25   C 0                stop: magazijn vol
       18   2T   8   X 0   E  =>        klaar
  0 => 19   2B  20 Z E 1                ledigplaats, want vertaalscan
       20   2S   0   X 0 B              magazijnwoord
       21   2B  13 Z E 2                schuifwijzer
       22   3P   0   SS  B
       23   2LS 127      A              isoleer symbool
       24   6S   9 Z E 0                berg symbool in DL
       25   1B   7       A P
       26 N 2B  15       A
       27   6B  13 Z E 2                nieuwe schuifwijzer
       28 N 2B   1       A
       29 N 4B  20 Z E 1                nieuwe ledigplaats
       30   2T   8   X 0   E  =>        klaar
  1 => 31   2A   0       A              voorbereiding
            DC D0
```

ZY1

```
     DA   0 Z Y 1       DI
 0   6A   9 Z E 1                QC:= 0
 1   6A  18 Z E 2                case RFS:= 0
 2   6A  21 Z E 2                voorraad NSS:= 0
 3   2S  15 Z E 2   P            voorbereiding voor prescan?
 4 Y 2A   1       A
 5 N 2A  18       A              zet strooisprong op
 6   6A  14 Z E 2                        voorbereiding geweest
 7 N 2T   0 Z Y 0 A     ->       klaar als vertaalscan
 8   2B  18 Z E 1                BIM
 9   0B   8       A
10   6B  19 Z E 1                vulplaats
11   6B  20 Z E 1                ledigplaats
12   2A   0       A
13   6A   0   X 0 B              clear eerste magazijnwoord
14   2A  22       A
15   6A  12 Z E 2                schuifwijzer voor vullen
16   2A  15       A
17   6A  13 Z E 2                schuifwijzer voor legen
18   2T   0 Z Y 0 A     =>       klaar met voorbereiding
     DC D0
```

```
THENELSE                                                        ZN0

aanroep                        6T 0 Z N0 1  =)  THENELSE

return with YES condition if THEN or ELSE on top of TLI


          DA   0 Z N 0       DI
     =)  0   2B  25 Z E 0
         1   2S  32767 X 0 B            S:= TLI[TLSC - 1]
         2   2LS 255      A             isoleer delimiter
         3 U 0LS 83       A Z           = then?
         4 N 0LS 84       A Z           of = else?
         5 N 2T   9   X 0   Z   ->      zo neen, klaar met THENELSE = false
5FZ0 =)  6   1B   2       A P
         7   6B  25 Z E 0   E           TLSC:= TLSC - 2; cond:= YES
         8   2B   0   X 0 B             gedumpte FLSC
         9   0B  12 Z E 0               + FLIB
        10   2A  24 Z E 0               RLSC
        11   6T   0 F U 0 0     =)      FFL met RLSC
        12   2S  18 Z E 0 A             adres van EFLA
        13   6T   0 Z U 0 0     =)      LTF voor EFLA
        14   2T   9   X 0   Z   =>      klaar met THENELSE = true
             DC D0
```

```
          DDEL  :=                                                        EZ0

                DA    0 E Z 0       DI
      =>  0     2T   11 E Z 2 A     =>       doe eerst RLA
13EZ2=>   1     6A    9 Z E 0                DL:= 128 (vertaling STORE)
          2     2A    1       A
          3     6A    0 Z E 0                OFLA:= 1
          4 U   2A    6 Z E 0   Z            VFLA = 0?
          5 N   2T   25 E Z 0 A     ->       zo neen, dan for clause
          6 U   2A   17 Z E 0   Z            SFLA = 0?
          7 N   2T   24 E Z 1 A     ->       zo neen, dan switch declaratie
          8 U   2A   18 Z E 0   Z            EFLA = 0?
          9 Y   6A   18 Z E 0                zo ja, dan EFLA:= 1
         10 N   4A    9 Z E 0                zo neen, dan DL:= 129 (STORE ALSO)
         11     2A    2       A
         12     6A    5 Z E 0                OH:= 2
         13 U   2A   16 Z E 0   Z            PFLA = 0?
         14 Y   2T   21 E Z 0 A     ->       zo ja, dan assignment aan variable
         15     4A    9 Z E 0                DL:= 130 of 131 (STP of STAP)
         16     2S    8 Z E 0                ID
         17     0P    8   SS
         18     2LS  31       A              isoleer BN
         19     6T    0 Z T 0 0     =)       FTL met BN uit ID
         20     2T   23 E Z 0 A     =>
   14 => 21     2A   29 Z E 0   Z            NFLA = 0? dwz geindiceerd
         22 N   6T    0 Z H 0 1     =)       zo neen, dan GAI
   20 -> 23     6T    0 Z W 0 0     =)       FTD
         24     2T    0 E L 0 A     =>       terug naar basiscyclus
    5 => 25     6A   18 Z E 0                FOR CLAUSE:  EFLA:= 1
         26     2A   29 Z E 0   Z            NFLA = 0? dwz geindiceerd?
         27 N   6T    0 Z H 0 1     =)       zo neen, dan GAI
         28     2A   20       A              OPC van FOR1
         29     2S    0       A
         30     6T    0 Z F 0 0     =)       FRL met FOR1
         31     2A    2       A              opc 2: referentie naar FLI
                DC D0
```

```
            DA    0 E Z 1        DI
         0  2S   26 Z E 0                 FLSC
         1  6S   11 Z E 0                 dumpen in FORC
         2  0S    8 Z K 0
         3  6T    0 Z F 0 0      =)       FRL met X2X  2T 'FLSC'
         4  2S    1       A
         5  4S   26 Z E 0                 FLSC:= FLSC + 1
         6  2B   10 Z E 0                 pak de in FORA gedumpte FLSC
         7  0B   12 Z E 0                 FLIB
         8  2A   24 Z E 0                 RLSC
         9  6T    0 F U 0 0      =)       FFL, dus FLI[FORA]:= RLSC
        10  2A    0       A
        11  2S    6 Z K 0
        12  6T    0 Z F 0 0      =)       FRL met 2A  0 A
        13  2S   26 Z E 0                 FLSC
        14  6S   10 Z E 0                 dumpen in FORA
        15  0S    5 Z K 0
        16  2A    2       A               opc 2: referentie naar FLI
        17  6T    0 Z F 0 0      =)       FRL met X2X  2B 'FLSC'
        18  2A    1       A
        19  4A   26 Z E 0                 FLSC:= FLSC + 1
        20  2A    9       A               OPC van ETMP
        21  2S    0       A
        22  6T    0 Z F 0 0      =)       FRL met ETMP
        23  2T    0 E L 0 A      =>       terug naar basiscyclus
7EZ0 => 24  2A    2       A               SWITCH DECLARATIE
        25  2S   26 Z E 0                 FLSC
        26  0S    8 Z K 0
        27  6T    0 Z F 0 0      =)       FRL met X2X  2T 'FLSC'
        28  2S   26 Z E 0
        29  6T    0 Z T 0 0      =)       FTL met FLSC
        30  2S    1       A
        31  4S   26 Z E 0                 FLSC:= FLSC + 1
            DC D0
```

EZ2

```
             DA   0 E Z 2       DI
          0  2S  22 Z E 0                NID
          1  6T   0 Z T 0 0     =)       FTL met NID
          2  2A   0       A
          3  6A   5 Z E 0                OH:= 0
          4  6T   0 Z W 0 0     =)       FTD
4EK4 ->   5  2A  160      A
          6  6A   9 Z E 0                DL:= 160 (switchkomma)
          7  2S  24 Z E 0                RLSC
          8  6T   0 Z T 0 0     =)       FTL met RLSC
          9  6T   0 Z W 0 0     =)       FTD met switchkomma
         10  2T   0 E L 0 A     =>       terug naar basiscyclus
0EZ0 => 11  6T   0 K N 0 2     =)       RLA
         12  2A  128      A
         13  2T   1 E Z 0 A     =>
             DC D0
```

```
   DDEL  [                                                                  EE0



          DA   0 E E 0        DI
=>  0     2A  18 Z E 0   Z              EFLA = 0?
    1 Y   6T   0 K N 0 2      =)        zo ja, dan RLA
    2     2A   1       A
    3     6A   0 Z E 0                  OFLA:= 1
    4     2A   0       A
    5     6A   5 Z E 0                  OH:= 0
    6     2S  18 Z E 0                  ga vlaggen dumpen in TLI:
    7     6T   0 Z T 0 0      =)        FTL met EFLA
    8     2S   1 Z E 0
    9     6T   0 Z T 0 0      =)        FTL met IFLA
   10     2S   4 Z E 0
   11     6T   0 Z T 0 0      =)        FTL met MFLA
   12     2S  19 Z E 0
   13     6T   0 Z T 0 0      =)        FTL met FFLA
   14     2S   2 Z E 0
   15     6T   0 Z T 0 0      =)        FTL met JFLA
   16     2S  22 Z E 0
   17     6T   0 Z T 0 0      =)        FTL met NID
   18     2A   1       A                ga vlaggen zetten:
   19     6A  18 Z E 0                  EFLA:= 1
   20     6A   1 Z E 0                  IFLA:= 1
   21     2A   0       A
   22     6A   4 Z E 0                  MFLA:= 0
   23     6T   0 Z W 0 0      =)        FTD met [
   24     2S   2 Z E 0   Z              JFLA = 0?
   25 Y   6T   0 Z H 0 1      =)        zo ja, dan GAI voor arraySTOFU
   26     2T   0 E L 0 A      =>        terug naar basiscyclus
          DC D0
```

```
        DDEL  ]                                                                    EF0

             DA   0 E F 0       DI
2 -> =>  0   6T   0 Z S 0 1     =)       OH:= 1; POP
         1   6T   0 Z N 0 1     =)       THENELSE?
         2 Y 2T   0 E F 0 A     ->       zo ja, dan herhaal
         3   2A   1       A
         4   5A  25 Z E 0                TLSC:= TLSC - 1, dwz gooi [ weg
         5   2A   1 Z E 0   Z            IFLA = 0?
         6 Y 2T  30 E F 0 A     ->       dan arraydeclaratie
         7   2S  22 Z E 0 A              ga gedumpte vlaggen ophalen:
         8   6T   0 Z U 0 0     =)       LTF voor NID
         9   2S   2 Z E 0 A
        10   6T   0 Z U 0 0     =)       LTF voor JFLA
        11   2S  19 Z E 0 A
        12   6T   0 Z U 0 0     =)       LTF voor FFLA
        13   2S   4 Z E 0 A
        14   6T   0 Z U 0 0     =)       LTF voor MFLA
        15   2S   1 Z E 0 A
        16   6T   0 Z U 0 0     =)       LTF voor IFLA
        17   2S  18 Z E 0 A
        18   6T   0 Z U 0 0     =)       LTF voor EFLA
        19   2A   2 Z E 0   Z            JFLA = 0?
        20   2S   0       A
        21   2A   1       A
        22 Y 6A  13 Z E 0                zo ja, dan AFLA:= 1
        23 N 6A  29 Z E 0                zo neen, dan NFLA:= 1
        24 Y 2A  56       A              OPC van IND
        25 N 2A  29       A              OPC van SSI
        26   6T   0 Z F 0 0     =)       FRL met IND of SSI
        27 Y 2T   0 E L 0 A     ->       zo ja, dan terug naar basiscyclus
        28   6T   0 Z Y 0 0     =)       RNS voor volgende delimiter
        29   2T   0 E F 2 A     =>       ga ID invullen en door naar DDEL
   6 -> 30   2S   2 Z K 0                ARRAY DECLARATIE
        31   0S   6 Z E 2
             DC D0
```

EF1

```
         DA   0 E F 1       DI
      0  2A   0       A
      1  6T   0 Z F 0 0     =)      FRL met 2S 'AIC' A
      2  2A  24 Z E 1               IBD
      3  0A  90       A             OPC van RSF
      4  2S   0       A
      5  6T   0 Z F 0 0     =)      FRL met RSF of ISF
      6  2A  30 Z K 0               ga ID opbouwen: d26,d25,d15
      7 U 2A  24 Z E 1   Z          real?
      8 N 0A   4 Z K 0              zo neen, voeg d19 toe
      9  6A   5 Z E 2               frame opgebouwd
     10  2B  30 Z E 0               NLSC
     11  0B  31 Z E 0               NLIB
25 -> 12  2A   5 Z E 2              frame
     13  0A  23 Z E 1               voeg PNLV als adres toe
     14  6A  32767 X 0 B            berg ID in naamlijst op
     15  2S  32766 X 0 B
     16  2LS  7       A Z           eenwoordsnaam?
     17 Y 1B   2      A
     18 N 1B   3      A
     19  2A   3 Z E 2               IC, de dimensie van het array
     20  0A   3       A
     21  2P   5   AA                hoog PNLV op met IC + 3 als
     22  4A  23 Z E 1                   plaatsreservering voor STOFU
     23  2A   1       A
     24  5A   6 Z E 2   P           AIC:= AIC - 1; AIC > 0?
     25 Y 2T  12 E F 1 A    ->      zo ja, nog meer ID's te maken
     26  6T   0 F T 0 2     =)      RND voor , of ;
     27  2A   9 Z E 0
     28  0LA 91       A Z           DL = ;?
     29 N 2T  20 K F 2 A    ->      zo neen, nog meer arrays van dit
     30  6A  18 Z E 0               EFLA:= 0                    type
     31  2T   0 E L 0 A     =>      terug naar basiscyclus
         DC D0
```

```
                                                                    EF2

             DA    0 E F 2        DI
29EF0=>  0   2B   22 Z E 0                 NID
         1   0B   31 Z E 0                 NLIB
         2   2S    0   X 0 B
         3   6S    8 Z E 0                 ID:= NLI[NID]
         4   2B    0       A
         5   6B   16 Z E 0                 PFLA:= 0
         6   2T    0 E H 0 A      =>       DDEL
             DC D0
```

```
DDEL    distribution on delimiter                                   EH0



         DA   0 E H 0       DI
=>  0    2B   9 Z E 0                DL
    1 U 1B  65       A P
    2 N 2T   0 E S 0 A      ->       als DL is + of -
    3 U 1B  68       A P
    4 N 2T   0 E T 0 A      ->       als * of / of div
    5 U 1B  69       A P
    6 N 2T   0 K T 0 A      ->       als **
    7 U 1B  75       A P
    8 N 2T   0 K K 0 A      ->       als < <= = >= > <>
    9 U 1B  80       A P
   10 N 2T   0 K L 0 A      ->       als not and or implies eqv
   11   2T  69-E H 0 B      =>       strooisprong
   12   0A   0 K R 0                 goto
   13   0A   0 E Y 0                 if
   14   0A   0 E N 0                 then
   15   0A   0 F Z 0                 else
   16   0A   0 F E 0                 for
   17   0A   0 F L 0                 do
   18   0A   0 E K 0                 ,
   19   0B   0   X 0                 .
   20   0B   0   X 0                 ten
   21   0A   0 F N 0                 :
   22   0A  13 F S 2                 ;
   23   0A   0 E Z 0                 :=
   24   0B   0   X 0                 spatie
   25   0A   0 F H 0                 step
   26   0A   0 F K 0                 until
   27   0A   0 F F 0                 while
   28   0B   0   X 0                 comment
   29   0A   0 E W 0                 (
   30   0A   0 E U 0                 )
   31   0A   0 E E 0                 [
        DC D0
```

EH1

```
     DA    0 E H 1        DI
 0   0A    0 E F 0                 ]
 1   0A    0 K S 0                 (*
 2   0B    0   X 0                 *)
 3   0A    0 L Z 0                 begin
 4   0A   13 F S 2                 end
 5   0A    0 K H 0                 own
 6   0A    0 K Z 0                 Boolean
 7   0A    0 K Z 0                 integer
 8   0A    0 K E 0                 real
 9   0A    0 K F 0                 array
10   0A    0 H E 0                 switch
11   0A    0 H Y 0                 procedure
12   0B    0   X 0                 string
13   0B    0   X 0                 label
14   0B    0   X 0                 value
     DC D0
```

```
DDEL  ,                                                        EK0



            DA   0 E K 0       DI
11EH0=>  0   2A   1       A
         1   6A   0 Z E 0                 OFLA:= 1
         2   2A   1 Z E 0   Z             IFLA = 0?
         3 Y 2T   8 E K 0 A     ->        dan geen subscriptscheider
   6 ->  4   6T   0 Z S 0 1     =)        OH:= 1; POP
         5   6T   0 Z N 0 1     =)        THENELSE?
         6 Y 2T   4 E K 0 A     ->        zo ja, dan herhaal
         7   2T   0 E L 0 A     =>        terug naar basiscyclus
   3 =>  8   2A   6 Z E 0   Z             VFLA = 0?
         9 Y 2T  30 E K 0 A     ->        dan geen scheider in for list
  12 -> 10   6T   0 Z S 0 1     =)        OH:= 1; POP
        11   6T   0 Z N 0 1     =)        THENELSE?
        12 Y 2T  10 E K 0 A     ->        zo ja, dan herhaal
1FL0 -> 13   2B  25 Z E 0
        14   2S  32767 X 0 B              TLI[TLSC - 1]
        15   2LS 255      A               isoleer operator
        16 U 0LS 85       A Z             is deze for?
        17 Y 2A  21       A               zo ja, dan OPC van FOR2
        18 Y 2T  24 E K 0 A     ->        en klaar met analyse
        19   1B   1       A               zo neen,
        20   6B  25 Z E 0                 TLSC:= TLSC - 1
        21 U 0LS 96       A Z             was het dan misschien while?
        22 Y 2A  23       A               zo ja, dan OPC van FOR4
        23 N 2A  26       A               zo neen, dan OPC van FOR7
  18 -> 24   2S   0       A
        25   6T   0 Z F 0 0     =)        FRL met FOR2, FOR4 of FOR7
        26   2A   9 Z E 0
        27   0LA 86       A Z             DL = do? dwz, kwamen we uit DDEL do?
        28 Y 2T   2 F L 0 A     ->        dan terug naar DDEL do
        29   2T   0 E L 0 A     =>        anders terug naar basiscyclus
   9 => 30   2A   4 Z E 0   Z             MFLA = 0?
        31 Y 2T  31 E K 3 A     ->        dan geen parameterscheider
            DC D0
```

EK1

```
            DA   0 E K 1       DI
1EU0 ->  0   2B  25 Z E 0                PARAMETERSCHEIDER
         1   2S  32767 X 0 B             TLI[TLSC - 1]
         2   2LS 255      A              isoleer delimiter
         3 U 0LS 87       A Z            is deze een , ?
8,22     4 Y 2T  15 E K 1 A     ->       mogelijk geen impl.subr.
24   -> 5    6T   0 Z S 0 1     =)       OH:= 1; POP
10EK4    6   6T   0 Z N 0 1     =)       THENELSE?
         7 N 6T   0 E R 0 1     =)       zo neen, dan DOT?
         8 Y 2T   5 E K 1 A     ->       zo ja, dan herhaal
         9   2S   9 Z K 0                d24 en d20 toevoegen aan
  17 -> 10   4S  32766 X 0 B             TLI[TLSC - 2], de PORD in opbouw
        11   2S   0       A
        12   2A  13       A              OPC van EIS
        13   6T   0 Z F 0 0     =)       FRL met EIS
        14   2T  14 E K 2 A     =>       volgende parameter voorbereiden
   4 => 15   2A  13 Z E 0   Z            AFLA = 0?
        16 N 2S  10 Z K 0                zo neen, met d24, d20 en d19
        17 N 2T  10 E K 1 A     ->       impl.subr. gaan afmaken
        18   2S  32766 X 0 B
        19   1S  24 Z E 0   Z            TLI[TLSC - 2] = RLSC?
        20 Y 3A  19 Z E 0                en ook
        21 Y 2LA  2 Z E 0   Z            not (FFLA = 0 and JFLA = 0)?
        22 N 2T   5 E K 1 A     ->       zo neen, dan impl.subr. afmaken
        23   2A  29 Z E 0   Z            NFLA = 0?
        24 Y 2T   5 E K 1 A     ->       zo ja, dan impl.subr. afmaken
        25   2T   5 E K 4 A     =>       test op standaardfunctie
16EK4=> 26   4P       AS                 S:= A (:= ID): construeer PORD
        27 U 2LA  0 Z K 0   Z            statisch?
        28 Y 2T   4 E K 2 A     ->       dan analyse voortzetten
        29   2LS 32767    A              isoleer adres uit ID
        30   1P   5   SS                 schuif BN in kop
        31   0S   3 Z K 0                voeg d16 toe (t oneven)
             DC D0
```

EK2

```
            DA   0 E K 2       DI
         0 U 2LA  3 Z K 0    Z          non-formeel?
         1 N 0S  12 Z K 0               zo neen, voeg d17 toe (t:= 3)
         2 N 2T  13 E K 2 A     ->      en klaar als actuele zelf formeel
         3   2T   8 E K 2 A     =>      zo ja, ga Q construeren
28EK1=>  4   2LS 11 Z K 0               handhaaf d25, d24 en het adres
         5   0LA 13 Z K 0               inverteer d25 in ID
         6 U 2LA 14 Z K 0   Z           d25 = d24 = 0? dwz, is het FLI?
         7 Y 0S  12 Z K 0               zo ja, voeg d17 toe (t:= 2)
   3 ->  8 U 2LA 29 Z K 0   Z           non-procedure?
         9 N 0S  15 Z K 0               zo neen, dan d20 toevoegen (Q:= 2)
        10 N 2T  13 E K 2 A     ->      en klaar met PORD
        11 U 2LA  4 Z K 0   Z           real?
        12 N 0S   4 Z K 0               zo neen, dan d19 toevoegen (Q:= 1)
2,10 -> 13   6S  32766 X 0 B            TLI[TLSC - 2]:= PORD
14EK1-> 14   2A  87       A
        15   1A   9 Z E 0   Z           DL = , ?
        16 Y 2T   8 E W 1 A     ->      zo ja, dan volgende parameter
        17   2A   0       A             AFLEVERING PORD'S AAN RLI
        18   6A  14 Z E 0               PSTA:= 0 (telling aantal parameters)
3EK3 -> 19   2S  15 Z E 0 A             adres PSTB
        20   6T   0 Z U 0 0     =)      LTF voor delimiter
        21   2A  15 Z E 0
        22   2LA 255      A             isoleer delimiter
        23   0LA 87       A Z           is deze een , ?
        24 N 2T   4 E K 3 A     ->      zo neen, dan laatste PORD gehad
        25   2A   1       A
        26   4A  14 Z E 0               PSTA:= PSTA + 1
        27   2S  15 Z E 0 A             adres PSTB
        28   6T   0 Z U 0 0     =)      LTF voor PORD
        29   2S  15 Z E 0
        30 U 2LS  3 Z K 0   Z           d16 = 0? dwz, t even?
        31   2A   0       A
            DC D0
```

EK3

```
            DA    0 E K 3       DI
          0 Y 0P    2   AS                 zo ja,
          1 Y 3P    2   SS                 schuif d26 en d25 als opc naar A
          2   6T    0 Z F 0 0     =)       FRL met PORD
          3   2T   19 E K 2 A     =>       volgende PORD gaan afleveren
24EK2=>   4   3B    1       A
          5   0B   25 Z E 0
          6   6B   25 Z E 0                TLSC:= TLSC - 1
          7   2B    0   X 0 B              pak de in TLI gedumpte FLSC
          8   0B   12 Z E 0                FLIB
          9   2A   24 Z E 0                RLSC
         10   6T    0 F U 0 0     =)       FFL, dus FLI[TLI[TLSC]]:= RLSC
         11   2A    0       A
         12   2S    6 Z K 0
         13   0S   14 Z E 0                PSTA
         14   6T    0 Z F 0 0     =)       FRL met 2A 'PSTA' A
         15   2A    1       A
         16   5A    7 Z E 0                BN:= BN - 1
         17   2S   19 Z E 0 A              ga gedumpte vlaggen ophalen:
         18   6T    0 Z U 0 0     =)       LTF voor FFLA
         19   2S   18 Z E 0 A
         20   6T    0 Z U 0 0     =)       LTF voor EFLA
         21   6T    0 Z L 0 0     =)       PTM
         22   2A    0       A
         23   6A   13 Z E 0                AFLA:= 0
         24   2S    4 Z E 0 A
         25   6T    0 Z U 0 0     =)       LTF voor MFLA
         26   2S    6 Z E 0 A
         27   6T    0 Z U 0 0     =)       LTF voor VFLA
         28   2S    1 Z E 0 A
         29   6T    0 Z U 0 0     =)       LTF voor IFLA
         30   2T    0 E L 0 A     =>       terug naar basiscyclus
31EK0=>  31   2A   17 Z E 0   Z            SFLA = 0? (dan array declaratie)
              DC D0
```

```
            DA   0 E K 4       DI
        0   6T   0 F R 0 2     =)       ETT
        1 Y 2T   0 E L 0 A     ->       en zo ja, dan terug naar basiscyclus
        2   2A   0       A              zo neen, dan scheider in switch list
        3   6A   5 Z E 0                OH:= 0
        4   2T   5 E Z 2 A     =>       verder samen met DDEL :=
25EK1=> 5   2A  22 Z E 0
        6   1A  25 Z E 2   P            NID > NLSCop? dwz, <> standaardftie?
        7 N 2A  98       A              zo neen,
        8 N 2S   0       A
        9 N 6T   0 Z F 0 0     =)       FRL met TFP
       10 N 2T   5 E K 1 A     ->       en klaar als standaardfunctie
       11   2A  16 Z E 0   Z            zo ja, is PFLA = 0?
       12 N 1A  19 Z E 0   Z            zo neen, is dan FFLA = 1?
       13 N 6T   0 L F 0 0     =)       TFO voor non-formele procedure
       14   2B  25 Z E 0                neem TLSC weer op
       15   2A   8 Z E 0                ID
       16   2T  26 E K 1 A     =>       ga PORD construeren
            DC D0
```

```
        basiscyclus                                                  EL0

                 DA   0 E L 0        DI
         =>  0   6T   0 F T 0 2      =)       RND
3HY1 ->  1   2A  29 Z E 0   Z                 NFLA = 0?
31KZ0    2 Y 6T  25 F W 1 0          =)       zo ja, clear vlaggen
         3 Y 2T   0 E H 0 A          ->       en weg naar DDEL
         4   2A   3 Z E 1   Z                 KFLA = 0?
         5 Y 6T   0 H Z 0 0          =)       LFN indien identifier
         6 N 6T   0 F W 0 0          =)       LFC indien constante
         7   2T   0 E H 0 A          =>       naar DDEL
             DC D0
```

```
DOT     do in TLI?                                                      ER0

aanroep                         6T 0 E R0 1  =)  DOT

return with YES condition if DO on top of TLI


     DA   0 E R 0       DI
=) 0   2B  25 Z E 0
   1   2S  32767 X 0 B            S:= TLI[TLSC - 1]
   2   2LS 255      A             isoleer delimiter
   3   0LS 86       A Z           = do?
   4 N 2T   9   X 0   Z           zo neen, dan klaar, DOT = false
   5   1B   5       A
   6   6B  25 Z E 0               TLSC:= TLSC - 5
   7   2A   2   X 0 B
   8   6A  30 Z E 0               NLSC:= TLI[TLSC + 2]
   9   2A   1       A
  10   5A   7 Z E 0               BN:= BN - 1
  11   2S   0   X 0 B             gedumpte RLSC
  12   2B   1   X 0 B             gedumpte FLSC
  13   0A  24 Z E 0               RLSC heersend
  14   0B  12 Z E 0               FLIB
  15   6T   0 F U 0 0     =)      FFL, dus FLI[TLI[TLSC]:= RLSC + 1
  16   0S   7 Z K 0
  17   2A   1       A P           opc1: relatief tov RLIB
  18   6T   0 Z F 0 0     =)      FRL met X1X 2T 'gedumpte RLSC' A
  19   2T   9   X 0   Z   =>      klaar met DOT = true
     DC D0
```

```
DDEL  + -                                                              ES0

         DA   0 E S 0       DI
   =>  0   2A   0 Z E 0   Z          OFLA = 0?
       1 N 2T   6 E S 0 A     ->     zo neen, dan + of - als teken
       2   2A   9       A
       3   6T   1 Z S 0 1     =)     OH:= 9; POP
       4   6T   0 Z W 0 0     =)     FTD
       5   2T   0 E L 0 A     =>     terug naar basiscyclus
 1 =>  6   2A  64       A
       7   1A   9 Z E 0   Z          DL = + ?
       8 N 2A  10       A            zo neen, dan
       9 N 6A   5 Z E 0              OH:= 10,
      10 N 2A  132      A
      11 N 6A   9 Z E 0              DL:= NEG,
      12 N 6T   0 Z W 0 0     =)     en FTD met NEG en OH
      13   2T   0 E L 0 A     =>     terug naar basiscyclus
           DC D0
```

```
        DDEL  * / div                                              ET0



                 DA   0 E T 0       DI
     =>  0   2A  10       A
1KT0 ->  1   6T   1 Z S 0 1     =)       OH:= 10; POP
3KK0     2   6T   0 Z W 0 0     =)       FTD
2KL0     3   2T   0 E L 0 A     =>       terug naar basiscyclus
             DC D0
```

```
     DDEL  (                                                          EW0

              DA    0 E W 0        DI
      =>  0   2A    1        A
          1   6A    0 Z E 0                OFLA:= 1
          2   2A   16 Z E 0    Z           PFLA = 0?
          3 N 2T   11 E W 0 A      ->      zo neen, dan procedurehaakje
17LH0->   4   2S    4 Z E 0                expressiehaakje:
          5   6T    0 Z T 0 0      =)      FTL met MFLA
          6   2A    0        A
          7   6A    4 Z E 0                MFLA:= 0
12EW1->   8   6A    5 Z E 0                OH:= 0
          9   6T    0 Z W 0 0      =)      FTD
         10   2T    0 E L 0 A      =>      terug naar basiscyclus
   3 =>  11   6T    0 L H 0 3      =)      PST
         12   2A    2        A             opc2: referentie naar FLI
         13   2S   26 Z E 0                FLSC
         14   0S    8 Z K 0
         15   6T    0 Z F 0 0      =)      FRL met X2X 2T 'FLSC'
         16   2S    1 Z E 0                ga vlaggen dumpen:
         17   6T    0 Z T 0 0      =)      FTL met IFLA
         18   2S    6 Z E 0
         19   6T    0 Z T 0 0      =)      FTL met VFLA
         20   2S    4 Z E 0
         21   6T    0 Z T 0 0      =)      FTL met MFLA
         22   2S   18 Z E 0
         23   6T    0 Z T 0 0      =)      FTL met EFLA
         24   2S   19 Z E 0
         25   6T    0 Z T 0 0      =)      FTL met FFLA
         26   2S   26 Z E 0
         27   6T    0 Z T 0 0      =)      FTL met FLSC
         28   2A    0        A             ga vlaggen zetten:
         29   6A    1 Z E 0                IFLA:= 0
         30   6A    6 Z E 0                VFLA:= 0
         31   2S    1        A
              DC D0
```

```
            DA   0 E W 1      DI
         0  6S   4 Z E 0               MFLA:= 1
         1  6S  18 Z E 0               EFLA:= 1
         2  4S  26 Z E 0               FLSC:= FLSC + 1
         3  6A   5 Z E 0               OH:= 0
         4  4S   7 Z E 0               BN:= BN + 1
         5  6T   0 Z W 0 0    =)       FTD
         6  2A  87       A
         7  6A   9 Z E 0               DL:= ,
16EK2->  8  2S  24 Z E 0
         9  6T   0 Z T 0 0    =)       FTL met RLSC
        10  2A   0       A
        11  6A  13 Z E 0               AFLA:= 0
        12  2T   8 E W 0 A    =>       verder als expressiehaakje
            DC D0
```

```
DDEL  )                                                              EU0


          DA   0 E U 0       DI
   =>  0    2A   4 Z E 0   Z            MFLA = 0?
       1 N 2T   0 E K 1 A     ->        zo neen, doe alsof parameterkomma
 4 ->  2    6T   0 Z S 0 1    =)        OH:= 1; POP
       3    6T   0 Z N 0 1    =)        THENELSE?
       4 Y 2T   2 E U 0 A     ->        zo ja, dan herhaal
       5    2A   1        A             verwijder ( uit TLI:
       6    5A  25 Z E 0                TLSC:= TLSC - 1
       7    2S   4 Z E 0 A              haal gedumpte vlag op:
       8    6T   0 Z U 0 0    =)        LTF voor MFLA
       9    2T   0 E L 0 A    =>        terug naar basiscyclus
          DC D0
```

DDEL if EY0

```
            DA   0 E Y 0       DI
     =>  0   2A  18 Z E 0   Z           EFLA = 0?
         1 Y 6T   0 K N 0 2     =)      zo ja, dan RLA
         2   2S  18 Z E 0
         3   6T   0 Z T 0 0     =)      FTL met EFLA
         4   2A   1       A
         5   6A  18 Z E 0               EFLA:= 1
19EN0->  6   2A   0       A
3KL0 ->  7   6A   5 Z E 0               OH:= 0
         8   6T   0 Z W 0 0     =)      FTD
         9   2A   1       A
        10   6A   0 Z E 0               OFLA:= 1
        11   2T   0 E L 0 A     =>      terug naar basiscyclus
            DC D0
```

```
      DDEL  then                                                          ENO



                DA    0 E N 0       DI
2->   =>  0     6T    0 Z S 0 1     =)       OH:= 1; POP
          1     6T    0 Z N 0 1     =)       THENELSE?
          2 Y   2T    0 E N 0 A     ->       zo ja, dan herhaal
          3     3B    1       A
          4     0B   25 Z E 0                verwijder if uit TLI:
          5     6B   25 Z E 0                TLSC:= TLSC - 1
          6     2A   32767 X 0 B
          7     6A   18 Z E 0                EFLA:= TLI[TLSC - 1]
          8     2A   30       A              OPC van CAC
          9     2S    0       A
         10     6T    0 Z F 0 0     =)       FRL met CAC
         11     2A    2       A              opc2: referentie naar FLI
         12     2S   26 Z E 0                FLSC
         13     0S   16 Z K 0
         14     6T    0 Z F 0 0     =)       FRL met X2X N 2T 'FLSC'
3FZ1 ->  15     2S   26 Z E 0
         16     6T    0 Z T 0 0     =)       FTL met FLSC
         17     2A    1       A
         18     4A   26 Z E 0                FLSC:= FLSC + 1
         19     2T    6 E Y 0 A     =>       verder samen met DDEL if
                DC D0
```

```
      DDEL  else                                                          FZ0



                 DA   0 F Z 0       DI
6->   =>  0      6T   0 Z S 0 1     =)       OH:= 1; POP
          1      2B  25 Z E 0
          2      2S  32767 X 0 B             S:= TLI[TLSC - 1]
          3      2LS 255      A              isoleer delimiter
          4 U    0LS 84       A Z            is deze een else?
          5 Y    6T   6 Z N 0 1     =)       zo ja, dan THENELSE
          6 Y    2T   0 F Z 0 A     ->       en herhaal
8,27 ->   7      6T   0 E R 0 1     =)       DOT?
          8 Y    2T   7 F Z 0 A     ->       zo ja, dan herhaal
          9      2B  25 Z E 0
         10      2S  32767 X 0 B             S:= TLI[TLSC - 1]
         11 U    0LS 161      A Z            blokbeginmarker op top TLI?
         12 N    2T  28 F Z 0 A     ->       zo neen, dan eenvoudig
         13      1B   3       A              ga eerst blok afronden:
         14      6B  25 Z E 0                TLSC:= TLSC - 3
         15      2S   1   X 0 B
         16      6S  30 Z E 0                NLSC:= TLI[TLSC + 1]
         17      2B   0   X 0 B              pak gedumpte FLSC
         18      0B  12 Z E 0                FLIB
         19      2A   1       A
         20      0A  24 Z E 0                RLSC
         21      6T   0 F U 0 0     =)       FFL, dus FLI[TLI[TLSC]]:= RLSC + 1
         22      2S   0       A
         23      2A  12       A              OPC van RET
         24      6T   0 Z F 0 0     =)       FRL met RET
         25      2A   1       A
         26      5A   7 Z E 0                BN:= BN - 1
         27      2T   7 F Z 0 A     =>       en herhaal DOT-test
  12 =>  28      2A   2       A              opc2: referentie naar FLI
         29      2S  26 Z E 0                FLSC
         30      0S   8 Z K 0
         31      6T   0 Z F 0 0     =)       FRL met X2X 2T 'FLSC'
                 DC D0
```

FZ1

```
    DA    0 F Z 1       DI
0   6T    0 Z N 0 1     =)        THENELSE (vindt then)
1   2A    1       A               behoud EFLA in TLI:
2   4A   25 Z E 0                 TLSC:= TLSC + 1
3   2T   15 E N 0 A     =>        verder samen met DDEL then
    DC D0
```

```
DDEL  for                                                         FE0



             DA   0 F E 0        DI
     =>  0   6T   0 K N 0 2      =)        RLA
         1   2A   2       A                opc2: referentie naar FLI
         2   2S  26 Z E 0                  FLSC
         3   0S   8 Z K 0
         4   6T   0 Z F 0 0      =)        FRL met X2X 2T 'FLSC'
         5   2A  26 Z E 0                  FLSC
         6   6A  10 Z E 0                  dumpen in FORA
         7   0A   1       A
         8   6A  26 Z E 0                  FLSC:= FLSC + 1
         9   2S  24 Z E 0
        10   6T   0 Z T 0 0      =)        FTL met RLSC
        11   2A   1       A
        12   6A   6 Z E 0                  VFLA:= 1
3FL1    13   4A   7 Z E 0                  BN:= BN + 1
4FF0 -> 14   2A   0       A
2KF3    15   6A   5 Z E 0                  OH:= 0
4LZ0    16   6T   0 Z W 0 0      =)        FTD
        17   2T   0 E L 0 A      =>        terug naar basiscyclus
             DC D0
```

```
        DDEL  while                                                    FF0



                DA   0 F F 0      DI
     =>  0      6T   0 F R 0 2    =)       ETT
         1      2A  22       A             OPC van FOR3
2KF0 ->  2      2S   0       A
         3      6T   0 Z F 0 0    =)       FRL met FOR3 of FOR6
         4      2T  14 F E 0 A    =>       naar einde DDEL for
                DC D0
```

DDEL step FH0

```
       DA   0 F H 0       DI
=> 0   6T   0 F R 0 2     =)       ETT
   1   2A  24       A              OPC van FOR5
   2   2S   0       A
   3   6T   0 Z F 0 0     =)       FRL met FOR5
   4   2T   0 E L 0 A     =>       terug naar basiscyclus
       DC D0
```

```
   DDEL  until                                                            FK0


         DA   0 F K 0       DI
=>  0    6T   0 F R 0 2     =)       ETT
    1    2A  25       A              OPC van FOR6
    2    2T   2 F F 0 A     =>       naar einde DDEL while
         DC D0
```

```
DDEL  do                                                        FL0


            DA   0 F L 0       DI
     => 0   6T   0 F R 0 2     =)       ETT
        1   2T  13 E K 0 A     =>       doe een stuk uit DDEL ,
28EK0=> 2   6A   6 Z E 0                VFLA:= 0 (einde for clause)
        3   2A   1       A              verwijder for uit TLI:
        4   5A  25 Z E 0                TLSC:= TLSC - 1
        5   2A   2       A              opc2: referentie naar FLI
        6   2S  26 Z E 0                FLSC
        7   0S  17 Z K 0
        8   6T   0 Z F 0 0     =)       FRL met X2X 2S 'FLSC'
        9   2S  26 Z E 0
       10   6T   0 Z T 0 0     =)       FTL met FLSC
       11   2A   1       A
       12   4A  26 Z E 0                FLSC:= FLSC + 1
       13   2A  27       A              OPC van FOR8
       14   2S   0       A
       15   6T   0 Z F 0 0     =)       FRL met FOR8
       16   2B  10 Z E 0                pak de in FORA gedumpte FLSC
       17   0B  12 Z E 0                FLIB
       18   2A  24 Z E 0                RLSC
       19   6T   0 F U 0 0     =)       FFL, dus FLI[FORA]:= RLSC
       20   2A  19       A              OPC van FOR0
       21   2S   0       A
       22   6T   0 Z F 0 0     =)       FRL met FOR0
       23   2A   1       A              opc1: relatief tov RLIB
       24   2B  25 Z E 0
       25   2S  32766 X 0 B             TLI[TLSC - 2]
       26   0S   7 Z K 0
       27   6T   0 Z F 0 0     =)       FRL met X1X 2T 'gedumpte RLSC' A
       28   2B  11 Z E 0                pak de in FORC gedumpte FLSC
       29   0B  12 Z E 0                FLIB
       30   2A  24 Z E 0                RLSC
       31   6T   0 F U 0 0     =)       FFL, dus FLI[FORC]:= RLSC
            DC D0
```

FL1

```
    DA   0 F L 1       DI
0   2A   0       A
1   6A  18 Z E 0                EFLA:= 0
2   6T   2 H W 0 0     =)       INB
3   2T  14 F E 0 A     =>       naar einde DDEL for
    DC D0
```

```
ETT      empty TLI through THENELSE                                FR0

aanroep                        6T 0 F R0 2  =)  ETT



        DA   0 F R 0       DI
  =)  0   2A  10   X 0
      1   6A   4 Z E 1               red link
      2   2A   1       A
      3   6A   0 Z E 0               OFLA:= 1
6 ->  4   6T   0 Z S 0 1     =)      OH:= 1; POP
      5   6T   0 Z N 0 1     =)      THENELSE?
      6 Y 2T   4 F R 0 A     ->      zo ja, dan herhaal
      7   2T   4 Z E 1   E   =>      klaar, terug via geredde link
          DC D0
```

```
        DDEL  ;        DDEL  end                                         FS0



             DA    0 F S 0       DI
20FS1=>  0   6T    0 F R 0 2     =)       ETT
2        1   6T    0 E R 0 1     =)       DOT?
         2 Y 2T    0 F S 0 A     ->       zo ja, dan herhaal
         3   2A   17 Z E 0   Z            SFLA = 0?
         4 Y 2T    2 F S 1 A     ->       dan niet einde van switchdeclaratie
         5   2A    0       A              ga switchdeclaratie afmaken:
         6   6A   17 Z E 0                SFLA:= 0
  17 ->  7   2B   25 Z E 0
         8   2S   32767 X 0 B             TLI[TLSC - 1]
         9 U 0LS 160       A Z            = switchkomma?
        10 N 2T   18 F S 0 A     ->       zo neen, dan laatste element gehad
        11   1B    2       A
        12   6B   25 Z E 0                TLSC:= TLSC - 2
        13   2S    0   X 0 B              TLI[TLSC]
        14   0S    7 Z K 0
        15   2A    1       A              opc1: relatief tov RLIB
        16   6T    0 Z F 0 0     =)       FRL met X1X 2T 'gedumpte RLSC' A
        17   2T    7 F S 0 A     =>       volgende sport van switchladder
  10 => 18   1B    1       A              verwijder := uit TLI:
        19   6B   25 Z E 0                TLSC:= TLSC - 1
        20   2S   22 Z E 0 A
        21   6T    0 Z U 0 0     =)       LTF voor NID
        22   6T    0 F Y 0 2     =)       LDEC
        23   2S   20 Z K 0
        24   2A    0       A
        25   6T    0 Z F 0 0     =)       FRL met 1T 16 X1
        26   2B   25 Z E 0
        27   1B    1       A
        28   6B   25 Z E 0                TLSC:= TLSC - 1
        29   2B    0   X 0 B              TLI[TLSC]
        30   0B   12 Z E 0                FLIB
        31   2A   24 Z E 0                RLSC
             DC D0
```

FS1

```
            DA   0 F S 1        DI
         0  6T   0 F U 0 0      =)      FFL, dus FLI[TLI[TLSC]]:= RLSC
         1  2T  10 F S 2 A      =>      ga EFLA op 0 zetten en testen
4FS0 =>  2  2B  25 Z E 0
         3  2S  32767 X 0 B             TLI[TLSC - 1]
         4 U 0LS 161      A Z           blokbeginmarker op top van TLI?
         5 N 2T  10 F S 2 A     ->      zo neen, ga dan EFLA op 0 zetten
         6  1B   3        A             ga eerst blok afronden:
         7  6B  25 Z E 0                TLSC:= TLSC - 3
         8  2S   1   X 0 B
         9  6S  30 Z E 0                NLSC:= TLI[TLSC + 1]
        10  2B   0   X 0 B              pak gedumpte FLSC
        11  0B  12 Z E 0                FLIB
        12  2A   1        A
        13  0A  24 Z E 0                RLSC
        14  6T   0 F U 0 0      =)      FFL, dus FLI[TLI[TLSC]]:= RLSC + 1
        15  2S   0        A
        16  2A  12        A             OPC van RET
        17  6T   0 Z F 0 0      =)      FRL met RET
        18  2A   1        A
        19  5A   7 Z E 0                BN:= BN - 1
        20  2T   0 F S 0 A      =>      en begin van voor af aan
12FS2=> 21  2S   9 Z E 0                DL
        22  0LS 105       A Z           = end?
        23 N 2T   0 E L 0 A     ->      zo neen, dan terug naar basiscyclus
        24  2A  25 Z E 0                verwijder begin uit TLI:
        25  1A   1        A
        26  6A  25 Z E 0                TLSC:= TLSC - 1
        27  1A   1        A
        28  1A   8 Z E 1   Z            TLSC = 1? (alleen nog BB in TLI?)
        29 Y 2T   0 K W 0 A     ->      zo ja, dan einde programma
3FS2 -> 30  6T   0 Z Y 0 0      =)      RNS
        31  2A   9 Z E 0                DL
            DC D0
```

FS2

```
              DA    0 F S 2       DI
         0 U 0LA 91        A Z          DL = ;?
         1 N 1A  84        A Z          of DL = else?
         2 N 1A  21        A Z          of DL = end?
         3 N 2T  30 F S 1 A     ->      zo neen, dan commentaar skippen
         4   2A   0        A            ga vlaggen zetten:
         5   6A   2 Z E 0               JFLA:= 0
         6   6A  16 Z E 0               PFLA:= 0
         7   6A  19 Z E 0               FFLA:= 0
         8   6A  29 Z E 0               NFLA:= 0
         9   2T   0 E H 0 A     =>      naar DDEL
1FS1 => 10   2S   0        A
5FS1    11   6S  18 Z E 0               EFLA:= 0
        12   2T  21 F S 1 A     =>      ga testen op end
DDEL => 13   6T   0 K N 0 2     =)      RLA
        14   2T   0 F S 0 A     =>      naar begin van deze DDEL
             DC D0
```

```
RND      read until next delimiter                              FT0

aanroep                           6T 0 F T0 2  =)  RND

NFLA = 0                          geen identifier of getal gelezen
NFLA = 1       KFLA = 0           identifier gelezen
               KFLA = 1           constante gelezen

          DA   0 F T 0       DI
    =)  0   6T   0 Z Y 0 0     =)       RNS
    =)  1   2S   1       A
        2   6S  29 Z E 0                NFLA:= 1
        3   2A   9 Z E 0                DL
        4 U 1A  63       A P
        5 U 1A   9       A E            verschillend van letter?
        6 Y 2T  15 F T 1 A     ->       zo ja, dan geen identifier
        7   2S   0       A
        8   6S   2 Z E 1                DFLA:= 0
        9   6S   3 Z E 1                KFLA:= 0
18 ->  10   1P   6   SA                 schuif symbool naar kop van S
       11 U 2LS  7       A Z            nog minder dan 5 symbolen?
       12   6S   1 Z E 1                INW
       13 N 2T  20 F T 0 A     ->       zo neen, dan dubbele naam
       14   6T   0 Z Y 0 0     =)       RNS
       15   2S   1 Z E 1                INW
       16   2A   9 Z E 0                DL
       17 U 1A  63       A P            geen letter of cijfer?
       18 N 2T  10 F T 0 A     ->       zo neen, dan voortgaan
       19   2T  22 F T 4 A     =>       zo ja, dan klaar met enkele naam
13 =>  20   2S   1       A
       21   6S   2 Z E 1                DFLA:= 1
       22   0A  18 Z K 0                d23 (als eindmarker)
       23   6A   0 Z E 1                FNW
31 ->  24   6T   0 Z Y 0 0     =)       RNS
       25   2A   0 Z E 1                FNW
       26   2S   9 Z E 0                DL
       27 U 1S  63       A P            verschillend van letter of cijfer?
       28 Y 2T  18 F T 4 A     ->       dan klaar met dubbele naam
       29   1P   6   SA    Z            aantal symbolen nog minder dan 9?
       30   6A   0 Z E 1                FNW
       31 Y 2T  24 F T 0 A     ->       zo ja, dan voortgaan
          DC D0
```

FT1

```
              DA    0 F T 1       DI
    3 ->  0   6T    0 Z Y 0 0     =)       RNS
          1   2S    9 Z E 0                DL
          2   1S   63       A P            verschillend van letter of cijfer?
          3 N 2T    0 F T 1 A     ->       anders overtollig symbool skippen
          4   2T   22 F T 4 A     =>       klaar met naam van 9 symbolen
      =)  5   6T    0 Z Y 0 0     =)       RNS     SUBROUTINE TEST-OP-CIJFER
          6   2A    9 Z E 0                DL
9FT5 ->   7 U 1LA 88        A Z            = .?
          8 Y 2S    1       A              zo ja, decimale punt gevonden,
          9 Y 6S    2 Z E 1                dus DFLA:= 1
         10 Y 2T    4 F T 2 A     ->       en terug naar assemblagecyclus
         11 U 1LA 89        A Z            DL = ten?
         12 Y 2T   10 F T 2 A     ->       zo ja, ga exponent lezen
         13 U 1A    9       A P            verschillend van cijfer?
         14   2T    9   X 0   Z   =>       terug, met behoud van conditie
6FT0 =>  15   6S    3 Z E 1                KFLA:= 1
         16   2B    0       A
         17   6B    0 Z E 1                FNW:= 0
         18   6B    1 Z E 1                INW:= 0
         19   6B    2 Z E 1                DFLA:= 0
         20   6B    4 Z E 1                ELSC:= 0
         21   6T    7 F T 5 1     =)       test op ten en doe subr. test-op-cijfer
         22 Y 2B    0       A              als DL <> cijfer of ten dan
         23 Y 6B   29 Z E 0                NFLA:= 0 en
         24 Y 2T   25 F T 4 A     ->       ga testen op true en false
5FT2 ->  25   2S    0 Z E 1                FNW     CYCLUS GETALASSEMBLAGE
         26   2LS 19 Z K 0    Z            < 2**22? dan bijvermenigvuldigen:
         27 Y 2S    1 Z E 1                INW
         28 Y 0X   10       A              AS:= 10 * INW + cijfer
         29 Y 6S    1 Z E 1                nieuwe INW
         30 Y 2S    0 Z E 1                FNW
         31 Y 0X   10       A              AS:= 10 * FNW + overloop uit INW
              DC D0
```

```
             DA   0 F T 2       DI
          0 Y 6S   0 Z E 1                nieuwe FNW
          1   3S   2 Z E 1                DFLA
          2 N 0S   1       A              aantal cijfers tellen:
          3   4S   4 Z E 1                ELSC:= ELSC - DFLA + 0 of 1
10FT1->   4   6T   5 F T 1 1    =)        subr. test-op-cijfer
          5 N 2T  25 F T 1 A    ->        als DL niet <> cijfer
          6   2S   2 Z E 1   Z            DFLA = 0?
          7 Y 2S   0 Z E 1   Z            and FNW = 0?
          8 Y 2T  22 F T 4 A    ->        zo ja, dan klaar met integer
          9   2T  27 F T 2 A    =>        zo neen, dan gaan floaten
12FT1=>  10   6T   5 F T 1 1    =)        subr. test-op-cijfer
         11 N 2T  16 F T 2 A    ->        als DL niet <> cijfer
         12   0LA 64       A Z            DL = +?
         13   6T   0 Z Y 0 0    =)        RNS voor eerste cijfer exponent
         14 Y 2A   9 Z E 0                zo ja, dan A:= + DL
         15 N 3A   9 Z E 0                zo neen, dan A:= - DL
  11 ->  16   6A   2 Z E 1                DFLA:= eerste cijfer exponent
         17   2T  23 F T 2 A    =>        ga volgende cijfers lezen
  24 =>  18   2S   2 Z E 1   P            DFLA    CYCLUS OPBOUW EXPONENT
         19 N 5P       AA
         20   0X  10       A Z            S:= 10 * DFLA + sign(DFLA) * cijfer
         21 N 7Y   3   C 0                en stop als dit naar A overloopt
         22   6S   2 Z E 1                nieuwe DFLA
  17 ->  23   6T   5 F T 1 1    =)        subr. test-op-cijfer
         24 N 2T  18 F T 2 A    ->        als DL niet <> cijfer
         25   2S   2 Z E 1                DFLA met
         26   4S   4 Z E 1                ELSC samen de complete exponent
   9 ->  27   3A   0 Z E 1                FNW     CONVERSIE NAAR FLOATING
         28   3S   1 Z E 1                INW
         29   6P       AS    Z            normeer; kop = 0?
17FT4->  30 Y 7S   2 Z E 1                zo ja, dan DFLA:= 0 voor integer 0
         31 Y 2T  22 F T 4 A    ->        en klaar
             DC DO
```

FT3

```
            DA   0 F T 3      DI
         0    1B  2100     A            2**11 + 52 (P9-karakteristiek)
         1    7B   2 Z E 1              bijdrage tot binaire karakteristiek
         2    2B   8       A            B:= 8
         3 U 2A    4 Z E 1   P          ELSC >= 0?
         4 N 2T   16 F T 3 A     ->     zo neen, dan decimale exponent negatief
         5    7A   0 Z E 1              FNW:= - kop
         6    2T  30 F T 3 A     =>
  19 =>  7 U 0A   15   D14 B P          NEGATIEVE DECIMALE EXPONENT
         8 N 3P    1   AS               halveer zo nodig
         9    0D  15   D14 B            en deel door 10**B
        10    7S   0 Z E 1              FNW:= - nieuwe kop
        11    0D  15   D14 B            deel de rest ook nog
        12    3A  23   D14 B            de met 10**B
        13 N 0A    1       A            corresponderende binaire exponent
        14    4A   2 Z E 1              bijtellen bij de binaire karakteristiek
        15    3A   0 Z E 1              FNW
   4 -> 16 U 0B    4 Z E 1   P          ELSC > - B?
        17 Y 3B    4 Z E 1   Z          zo ja, dan B:= - ELSC; B = 0?
        18 N 4B    4 Z E 1              zo neen, dan ELSC:= ELSC + B
        19 N 2T    7 F T 3 A     ->     en verder gaan delen
        20    2T   3 F T 4 A     =>     reductie voltooid
1FT4 => 21    2X  15   D14 B            POSITIEVE DECIMALE EXPONENT
        22    3S   0 Z E 1              A:= 10**B * (- staart)
        23    0X  15   D14 B            AS:= 10**B * (-kop) + A
        24    0P   1   AS    P
        25 Y 1P    1   AS               zo mogelijk nog verdubbelen
        26    7A   0 Z E 1              FNW:= - nieuwe kop
        27    2A  23   D14 B            de met 10**B
        28 N 1A    1       A            corresponderende binaire exponent
        29    4A   2 Z E 1              bijtellen bij de binaire karakteristiek
   6 -> 30 U 1B    4 Z E 1   P          ELSC < B?
        31 Y 2B    4 Z E 1   Z          zo ja, dan B:= ELSC; B = 0?
            DC D0
```

FT4

```
            DA   0 F T 4        DI
          0 N 5B   4 Z E 1               zo neen, dan ELSC:= ELSC - B
          1 N 2T  21 F T 3 A     ->      en verder gaan vermenigvuldigen
          2   3A   0 Z E 1               FNW
20FT3->   3   1S  2048    A P            AFRONDING
          4 Y 3S   0      A              als staart overloopt dan
          5 Y 1A   1      A P            carry naar kop
          6 Y 1P   1   AA                zo nodig deze halveren
          7   7A   0 Z E 1               FNW:= voltooide kop
          8   5P       SS
          9   3LS 4095    A              in staart plaats maken
         10   6S   1 Z E 1               INW:= staart
         11   2S   2 Z E 1               binaire karakteristiek
         12 Y 0S   1      A
         13 U 3LS 4095    A Z            tussen - 4096 en + 4096?
         14 N 7Y   4   C 0               zo neen, dan overschrijding capaciteit
         15   4S   1 Z E 1               bijtellen bij staart in INW
         16   3S   1      A
         17   2T  30 F T 2 A     =>      DFLA op 1 gaan zetten en klaar
28FT0=>  18   3S   0      A              als naam <= 9 symbolen dan
20       19   1P   6   SA   P            'loos' bijschuiven
         20 Y 2T  18 F T 4 A     ->      zo nodig herhalen
         21   6A   0 Z E 1               FNW
19FT0->  22   2A   0      A
6FT5     23   6A   0 Z E 0               OFLA:= 0
         24   2T  10   X 0   E   =>      klaar
24FT1=>  25   2A   9 Z E 0               DL
         26 U 1A  117     A Z            = false?
         27 Y 2S   1      A
         28 N 2S   0      A
         29 U 1A  115     A E            of DL = true?
         30 Y 2T  10   X 0   E   ->      klaar als noch true noch false
         31   6S   1 Z E 1               INW:= 0 of 1
            DC D0
```

FT5

```
          DA   0 F T 5       DI
       0  2A   0       A
       1  6A   2 Z E 1                DFLA:= 0
       2  2A   1       A
       3  6A   3 Z E 1                KFLA:= 1
       4  6A  29 Z E 0                NFLA:= 1
       5  6T   0 Z Y 0 0     =)       RNS voor delimiter na true of false
       6  2T  22 F T 4 A     =>       klaar
21FT1=)  7 U 1LA 89       A Z         DL = ten?
       8 Y 6S   1 Z E 1               zo ja, maak numeriek gedeelte = 1
       9  2T   7 F T 1 A     =>       door naar test-op-cijfer
          DC D0
```

```
        LFC      look for constant                                       FW0

        aanroep                          6T 0 F W0 0  =)  LFC



             DA   0 F W 0       DI
     =)  0   2A  31 Z E 0               NLIB
         1   1A   2 Z E 1               DFLA
         2   1A  27 Z E 0               KLSC
         3   1A  28 Z E 0   P           KLIB + KLSC + DFLA < NLIB?
         4 N 2T  10 F W 1 A     ->      zo neen, dan NLI opschuiven
24FW1->  5   2B  27 Z E 0               KLSC
         6   0B  28 Z E 0               KLIB
         7   2S   1 Z E 1               INW
         8   2A   2 Z E 1   Z           DFLA = 0?
         9 N 2T  18 F W 0 A     ->      zo neen, dan floating getal
        10   6S   0   X 0 B             KLI[KLSC]:= integer
        11   2B  28 Z E 0               KLIB
  14 -> 12 U 0LS  0   X 0 B Z           CYCLUS ZOEK INTEGER
  16 -> 13 N 0B   1       A
        14 N 2T  12 F W 0 A     ->      als niet + 0 of - 0 dan volgende
        15 U 2A   1       A E           + 0?
        16 N 2T  13 F W 0 A     ->      zo niet, dan slechts complement
        17   2T  31 F W 0 A     =>      integer gevonden
   9 => 18   2A   0 Z E 1               FNW
        19   6A   0   X 0 B             KLI[KLSC]:= kop
        20   6S   1   X 0 B             KLI[KLSC + 1]:= staart
        21   2B  28 Z E 0               KLIB
  24 -> 22 U 0LA  0   X 0 B Z           CYCLUS ZOEK FLOATING
26,28-> 23 N 0B   1       A             als kop niet klopt
 30     24 N 2T  22 F W 0 A     ->      dan volgende
        25 U 2A   1       A E           + 0?
        26 N 2T  23 F W 0 A     ->      zo niet, dan slechts complement
        27 U 0LS  1   X 0 B Z           klopt ook de staart?
        28 N 2T  23 F W 0 A     ->      zo neen, dan volgende
        29 U 2A   1       A E           + 0?
        30 N 2T  23 F W 0 A     ->      zo niet, dan slechts complement
  17 -> 31   5P       BS
             DC D0
```

FW1

```
            DA    0 F W 1       DI
        0   0S   28 Z E 0
        1 U 0S   27 Z E 0   Z            KLSC teruggevonden?
        2 Y 2B    1       A              zo ja,
        3 Y 0B    2 Z E 1                dan nog niet eerder ontmoet,
        4 Y 4B   27 Z E 0                dus KLSC:= KLSC + DFLA + 1
        5   2A    2 Z E 1   Z            CONSTRUCTIE ID
        6 Y 3LS   4 Z K 0                als DFLA = 0 dan d19 toevoegen
        7   3LS  14 Z K 0                d25, d24 toevoegen als opc3
        8   7S    8 Z E 0                berg ID
        9   2T   25 F W 1 A     =>       ga vlaggen zetten
4FW0 => 10  2B   30 Z E 0                OPSCHUIVEN VAN NLI
       11   6B    0   X 0                aantal:= NLSC
       12   0B   31 Z E 0                NLIB
       13   5P       BS
       14   1S   16       A
       15   0S   21 Z E 0   P            NLIB + NLSC + 16 < PLIB?
       16 N 7Y    5   C 0                zo neen, stop: NLI schuift in PLI
       17   2T   21 F W 1 A     =>
  21 => 18  1B    1       A              opschuifcyclus:
       19   2S    0   X 0 B              16 plaatsen
       20   6S   16   X 0 B              omhoog
  17 -> 21  4T   18 F W 1 0 E   ->
       22   2S   16       A
       23   4S   31 Z E 0                NLIB:= NLIB + 16
       24   2T    5 F W 0 A     =>       klaar met schuiven
   9 => 25  2A    0       A              zet vlaggen
       26   6A    2 Z E 0                JFLA:= 0
       27   6A   16 Z E 0                PFLA:= 0
       28   6A   19 Z E 0                FFLA:= 0
       29   2T    8   X 0   E   =>       klaar
            DC D0
```

```
FFL      fill future list                                          FU0

aanroep                    6T 0 F U0 0  =)  FFL

functie                    FLI[B]:= A


          DA   0 F U 0       DI
21=> =)  0 U 5B  28 Z E 0   P        B < KLIB?
         1 Y 6A   0   X 0 B          zo ja, dan FLI[B]:= A
         2 Y 2T   8   X 0   E   ->   en klaar
         3   6B   1   X 1            OPSCHUIVEN VAN KLI EN NLI
         4   6A   0   X 1            red A en B
         5   2B  30 Z E 0            NLSC
         6   0B  31 Z E 0            NLIB
         7   5P       BA
         8   1A  16       A
         9 U 0A  21 Z E 0   P        NLIB + NLSC + 16 < PLIB?
        10 N 7Y   6   C 0            zo neen, stop: schuiven in PLI
        11   0A  16       A
        12   0A  28 Z E 0            KLIB
        13   7A   0   X 0            aantal:= NLIB + NLSC - KLIB
        14   2A  16       A
        15   4A  28 Z E 0            KLIB:= KLIB + 16
        16   4A  31 Z E 0            NLIB:= NLIB + 16
  20 -> 17   1B   1       A          opschuifcyclus
        18   2A   0   X 0 B          16 plaatsen
        19   6A  16   X 0 B          omhoog
        20   4T  17 F U 0 0 P   ->
        21   2A   0   X 1            herstel A en B
        22   2B   1   X 1
        23   2T   0 F U 0 A     =>   en opnieuw proberen
             DC D0
```

```
LDEC     label declaration                                            FY0

aanroep                           6T 0 F Y0 2  =)  LDEC



          DA   0 F Y 0       DI
   =)  0   2B  22 Z E 0                NID
       1   0B  31 Z E 0                NLIB
       2   2A   0   X 0 B              ID uit NLI
       3 U 2LA  0 Z K 0   Z            d15 = 0?
       4 N 2T  10 F Y 0 A     ->       zo neen, dan first occurrence
       5   4P       AB
       6   0B  12 Z E 0                FLIB
       7   2A  24 Z E 0                RLSC
       8   6T   0 F U 0 0     =)       FFL, dus FLI[FLSC uit ID]:= RLSC
       9   2T  15 F Y 0 A     =>       ga labelnaam typen
  4 => 10   3LA  0 Z K 0               d15:= 0
      11   3LA 32767    A              maak plaats voor adres
      12   0A  24 Z E 0                RLSC als adres
      13   0A  21 Z K 0                d24 als codering toevoegen
      14   6A   0   X 0 B              ID in NLI opbergen
  9 -> 15   2S  11   X 4 A
      16   6S  11 Z E 1                SHIFT:= undefined
      17   6T   0 H R 0 0     =)       FOB6 met TWNR
      18   2B  22 Z E 0                NID
      19   0B  31 Z E 0                NLIB
      20   2A  32767 X 0 B             INW uit NLI
      21 U 2LA  7       A Z            eenwoordsnaam?
      22 N 2T   2 F Y 1 A     ->       zo neen dan dubbele naam typen
      23   2B   4       A              hoogstens 4 letters of cijfers
      24   1P   3   AA
 28 -> 25 U 2LA 63       A Z           'letter' = loos?
      26 Y 1B   1       A              zo ja,
      27 Y 3P   6   AA                 dan overslaan
      28 Y 2T  25 F Y 0 A     ->       en herhalen
0FY1 -> 29   6T   0 H S 0 1     =)     OCT
      30   1B   1       A P            nog meer letters?
      31 Y 3P   6   AA                 zo ja,
          DC D0
```

FY1

```
            DA   0 F Y 1        DI
         0 Y 2T  29 F Y 0 A     ->      dan herhalen
         1   2T  10 F Y 1 A     =>      ga 32-tallig adres typen
22FY0=>  2   2S  32766 X 0 B            FNW uit NLI
         3   1P   3   SS
         4   0P   3   SA                stel beginletter samen
         5   2B   9       A             hoogstens 9 letters of cijfers
   9 ->  6   6T   0 H S 0 1     =)      OCT
         7   1B   1       A P           nog meer letters?
         8 Y 1P   6   SA                zo ja,
         9 Y 2T   6 F Y 1 A     ->      dan herhalen
   1 -> 10   2S  10   X 4 A
        11   6T   0 H R 0 0     =)      FOB6 met Tabulatie
        12   2B   3       A             3 groepen van 2 cijfers
        13   2S  24 Z E 0               RLSC gaan herleiden
        14   2T  30 F Y 1 A     =>
  28 => 15   2S  56   X 4 A
        16   6T   0 H R 0 0     =)      FOB6 met spatie
        17   2S  12 Z E 1               LDECA
  31 -> 18   2A   0       A             isoleer 32-tallige eenheid
        19   1P  10   SA
        20   1P  12   AA
        21   6A  12 Z E 1               LDECA:= rest
        22   2A   0       A
        23   0D  10       A
        24   0P  27   SA
        25   6T   0 H S 0 1     =)      OCT met eerste cijfer
        26   4P       SA
        27   6T   0 H S 0 1     =)      OCT met tweede cijfer
        28   4T  15 F Y 1 0 P   ->
        29   2T  10   X 0   E   =>      klaar
  14 => 30   6B   0   X 0               aantal:= 3
        31   2T  18 F Y 1 A     =>
             DC D0
```

```
DDEL  :                                                                  FN0

             DA   0 F N 0        DI
     =>  0   2A   2 Z E 0   Z             JFLA = 0?
         1 N 2T   6 F N 0 A      ->       zo neen, dan label gevonden
         2   2A   1       A               zo ja, dan arraydeclaratie:
         3   4A   3 Z E 2                 IC:= IC + 1
         4   6T   0 F R 0 2      =)       ETT
         5   2T   0 E L 0 A      =>       terug naar basiscyclus
   1 =>  6   6T   0 K N 0 2      =)       RLA
         7   6T   0 F Y 0 2      =)       LDEC
         8   2T   0 E L 0 A      =>       terug naar basiscyclus
             DC D0
```

```
LFN      look for name                                                HZ0

aanroep                          6T 0 H Z0 0  =)  LFN



          DA   0 H Z 0       DI
   =)  0   2B  30 Z E 0             NLSC
       1   0B  31 Z E 0             NLIB
13 ->  2   2A  32766 X 0 B          INW uit NLI
       3 U 1A   1 Z E 1   Z         klopt INW?
       4 N 2T   9 H Z 0 A     ->    zo neen, dan volgende proberen
       5 U 2LA  7       A Z         enkelwoords naam?
       6 N 2S  32765 X 0 B          zo neen,
       7 N 1S   0 Z E 1   Z         klopt dan ook FNW?
       8 Y 2T  15 H Z 0 A     ->    zo ja, dan naam gevonden
 4 ->  9   2LA  7       A Z         enkelwoords naam?
      10 Y 1B   2       A
      11 N 1B   3       A
      12 U 1B  31 Z E 0   P         nog in NLI?
      13 Y 2T   2 H Z 0 A     ->    zo ja, dan nog eens proberen
      14   2B   2 Z E 1 A Z         adres van DFLA
 8 -> 15 Y 1B   1       A
      16 Y 2A   0   X 0 B           ID uit NLI
      17   1B  31 Z E 0             NLIB
      18   6B  22 Z E 0             NID
      19 N 2T  31 H Z 0 A     ->    als niet naam-in-naamlijst
      20   6A   8 Z E 0             berg ID
      21   0P   9   SA              zet vlaggen:
      22   2LS  1       A
      23   6S  16 Z E 0   Z         PFLA:= d18 van ID
      24 N 2S   0       A
      25   0P   1   SA
      26   6S   2 Z E 0   Z         JFLA:= d17 van ID
      27 N 2S   0       A
      28   0P   1   SA
      29   6S  19 Z E 0             FFLA:= d16 van ID
      30   2T   8   X 0   E   =>    klaar
19 => 31   2A  12       A           NAAM NIET IN NAAMLIJST
          DC D0
```

HZ1

```
         DA   0 H Z 1      DI
      0  6A  26   X 0            klasse 6 in neutrale toestand
      1  7A   2 Z E 2            typ-magazijn leeg
      2  7A  17 Z E 1            typen imperatief
      3  2S  11   X 4 A
      4  6T   0 H R 0 0    =)    FOB6 met TWNR
      5  2A   0       A
      6  6A  24 Z E 0            RLSC:= 0
      7  6T  15 F Y 0 2    =)    LDEC voor typen van naam
 9 -> 8  6T   5   D 1 0    =)    TPA?
      9 Y 1T  2       A    ->    wacht dan op voltooiing
     10  7Y   7   C 0            stop: naam niet in NLI
         DC D0
```

```
DDEL  switch                                                    HE0

        DA   0 H E 0        DI
=>  0   6T   0 K N 0 2      =)       RLA
    1   2A   1       A
    2   6A  17 Z E 0                 SFLA:= 1
    3   6T   0 H U 0 1      =)       NBD
    4   2T   0 E L 0 A      =>       terug naar basiscyclus
        DC D0
```

```
        FPL     fill prescan list                                               HF0

        aanroep                           6T 0 H F0 0  =)  FPL met label of switch

                                          6T 2 H F0 0  =)  FPL met procedure-naam


             DA   0 H F 0      DI
     =)  0   2S   0       A
         1   2T   3 H F 0 A    =>
     =)  2   2S   1       A
   1 ->  3   0S   5 Z E 1                 BC
         4   0S   5 Z E 1
         5   6S   0   X 0                 aantal:= 2 * BC + 0 of 1
         6   2A   2 Z E 1                 DFLA
         7   0A   1       A
         8   2S  21 Z E 0 A               adres PLIB
  12 ->  9   4P      SB                   CYCLUS VERLAAG ADRESSEN IN
        10   2S   0   X 0 B                               PLI-KETTING
        11   5A   0   X 0 B
        12   4T   9 H F 0 0 E  ->
        13   6S   0   X 1                 S bevat nu het adres van het
        14   1S  21 Z E 0                 PLIB            laatste nog te
        15   1S   2 Z E 1                 DFLA            verschuiven woord
        16   6S   0   X 0                 aantal
        17   2B  21 Z E 0                 PLIB (is al afgelaagd)
        18 U 2A   2 Z E 1   Z             DFLA = 0?
        19 Y 2T  24 H F 0 A    ->         zo ja, dan 1 plaats verschuiven
        20   2T  31 H F 0 A    =>         zo neen, dan 2 plaatsen verschuiven
  24 => 21   2S   1   X 0 B               CYCLUS VERSCHUIF OVER 1 PLAATS
        22   6S   0   X 0 B
        23   0B   1       A
  19 -> 24   4T  21 H F 0 0 E  ->
2HF1 -> 25   2S   1 Z E 1                 INW
        26   6S   0   X 0 B               in PLI opnemen
        27   2T   8   X 0   E  =>         klaar
  31 => 28   2S   2   X 0 B               CYCLUS VERSCHUIF OVER 2 PLAATSEN
        29   6S   0   X 0 B
        30   0B   1       A
  20 -> 31   4T  28 H F 0 0 E  ->
             DC D0
```

HF1

```
    DA   0 H F 1       DI
0   2S   0 Z E 1                FNW
1   6S   1   X 0 B              in PLI opnemen
2   2T  25 H F 0 A     =>       ga INW in PLI opnemen
    DC D0
```

```
APL      augment prescan list                                    HH0

aanroep                          6T 0 H H0 0  =)  APL



        DA   0 H H 0       DI
=)  0   2A   1       A
    1   6A   2 Z E 1               DFLA:= 1
    2   2A   6 Z E 1               PLIE
    3   6A   1 Z E 1               INW:= PLIE
    4   1A   1       A
    5   6A   0 Z E 1               FNW:= PLIE - 1
    6   2T   0 H F 0 A     =>      door naar FPL met [PLIE,PLIE + 1]
        DC D0
```

```
       PSP      prescan program                                        HK0

       veronderstelling                DL = 0-de begin



            DA   0 H K 0       DI
8LE0 =>  0  2B   6 Z E 1               PLIE
         1  6B  21 Z E 0               PLIB:= PLIE
         2  1B   1       A
         3  6B   1   X 0 B             PLI[PLIE]:= PLIE - 1
         4  2A   8 Z E 1               TLIB
         5  6A  25 Z E 0               TLSC:= 0
         6  3S   0       A
         7  7S   5 Z E 1               BC:= 0
         8  7S   7 Z E 1               MBC:= 0
         9  7S   9 Z E 1               QC:= 0
        10  7S  26 Z E 2               RHT:= 0
        11  7S  27 Z E 2               VHT:= 0
        12  2S   9 Z E 0               DL, hopelijk een begin
        13  6T   0 Z T 0 0     =)      FTL met DL
        14  6T   0 H H 0 0     =)      APL
     -> 15  2A   0       A
        16  6A  10 Z E 1               BFLA:= 0
     -> 17  6T   0 F T 0 2     =)      RND
6HK3 -> 18  2S   9 Z E 0               DL
0HK3 -> 19 U 1S  84      A P           voor 'te kleine' delimiter
        20 N 2T  15 H K 0 A    ->      geen interesse
        21 U 1S  85      A Z
        22 Y 2T  14 H K 2 A    ->      als DL = for
        23 U 1S  89      A P           voor do of , of . of ten
        24 N 2T  15 H K 0 A    ->      geen interesse
        25 U 1S  90      A Z
        26 Y 2T  12 H K 2 A    ->      als DL = :
        27 U 1S  91      A Z
        28 Y 2T  11 H K 3 A    ->      als DL = ;
        29 U 1S  97      A P           voor := of step of until of while
        30 N 2T  15 H K 0 A    ->      of comment geen interesse
        31 U 1S  99      A P
            DC D0
```

HK1

```
            DA    0 H K 1        DI
         0 N 2T  25 H K 3 A      ->        als DL = ( of )
         1 U 1S  101      A P
         2 N 2T  30 H K 3 A      ->        als DL = [ of ]
         3 U 1S  102      A Z
         4 Y 2T  17 H K 2 A      ->        als DL is (*
         5 U 1S  104      A Z
         6 Y 2T  25 H K 2 A      ->        als DL = begin
         7 U 1S  105      A Z
         8 Y 2T  11 H K 3 A      ->        als DL = end
         9   2A   1       A E              voor *)
        10 N 2T  15 H K 0 A      ->        geen interesse
        11 U 1S  111      A Z
        12 Y 2T  29 H K 1 A      ->        als DL = switch
  20 -> 13 U 1S  112      A Z
        14 Y 2T  22 H K 1 A      ->        als DL = procedure
        15 U 1S  117      A P
        16 Y 7Y   8   C 0                  stop als DL ontoelaatbaar
23,30-> 17   6T   0 F T 0 2      =)        RND
  28 -> 18   2S   9 Z E 0                  DL      skip declaraties en
1HK2    19 U 1S  91       A Z                            specificaties
        20 N 2T  13 H K 1 A      ->        als DL niet ; dan skippen
        21   2T  17 H K 0 A      =>        prescan vervolgen
  14 => 22 U 2A  10 Z E 1   Z              BFLA = 0?               PROCEDURE
        23 N 2T  17 H K 1 A      ->        zo neen, dan specificatie: skip
        24   6A  10 Z E 1                  BFLA:= 1
        25   6T   0 F T 0 2      =)        RND voor procedure identifier
        26   6T   2 H F 0 0      =)        FPL
        27   6T   2 H K 2 1      =)        blokintroductie voor body
        28   2T  18 H K 1 A      =>        ga formele parameters skippen
  12 => 29 U 2A  10 Z E 1   Z              BFLA = 0?               SWITCH
        30 N 2T  17 H K 1 A      ->        zo neen, dan specificatie: skip
        31   6T   0 F T 0 2      =)        RND voor switch identifier
            DC D0
```

```
             DA   0 H K 2       DI
          0  6T   0 H F 0 0     =)      FPL
          1  2T  18 H K 1 A     =>      ga switch list skippen
      =)  2  2S   5 Z E 1               SUBROUTINE BLOKINTRODUCTIE
          3  6T   0 Z T 0 0     =)      FTL met BC
          4  3S   0       A
          5  6T   0 Z T 0 0     =)      FTL met blokbeginmarker
          6  2S   7 Z E 1               MBC
          7  0S   1       A
          8  6S   7 Z E 1               MBC:=
          9  6S   5 Z E 1                      BC:= MBC + 1
         10  6T   0 H H 0 0     =)      APL
         11  2T   9   X 0   E   =>      link
26HK0=>  12  6T   0 H F 0 0     =)      FPL met label identifier
         13  2T  17 H K 0 A     =>      vervolg prescan
22HK0=>  14  6T   2 H K 2 1     =)      blokintroductie voor for-blok
         15  2T  15 H K 0 A     =>      vervolg prescan met BFLA = 0
  23 =>  16  6T   0 Z Y 0 0     =)      RNS voor volgend stringsymbool
4HK1 ->  17  2S   9 Z E 0               DL
         18 U 1S  102     A Z           (*?
         19  2A   1       A
         20 Y 4A   9 Z E 1              zo ja, dan QC:= QC + 1
         21 U 1S  103     A Z           *)?
         22 Y 5A   9 Z E 1   Z          zo ja, dan QC:= QC - 1
         23 N 2T  16 H K 2 A    ->      als QC niet 0 dan herhalen
         24  2T  17 H K 0 A     =>      anders prescan voortzetten
6HK1 =>  25  6T   0 Z T 0 0     =)      FTL met begin          BEGIN
         26 U 2A  10 Z E 1   Z          BFLA = 0?
         27 N 2T  15 H K 0 A    ->      zo neen, prescan vervolgen met
         28  6T   0 F T 0 2     =)      RND                  BFLA = 0
         29  2S   9 Z E 0               DL
         30 U 1S  105     A P
         31 U 1S  112     A E           verschillend van declarator?
             DC D0
```

HK3

```
             DA    0 H K 3       DI
         0 Y 2T   19 H K 0 A     ->       dan geen nieuw blok
         1   3B    1       A              schrap begin uit TLI:
         2   4B   25 Z E 0                TLSC:= TLSC - 1
         3   6T    2 H K 2 1     =)       blokintroductie wegens declaratie
         4   2S   104      A              voeg begin weer toe:
         5   6T    0 Z T 0 0     =)       FTL met begin
         6   2T   18 H K 0 A     =>       zet prescan voort
  13 =>  7   1B    2       A              uitluiden van blok:
         8   6B   25 Z E 0                TLSC:= TLSC - 2
         9   2A    0   X 0 B
        10   6A    5 Z E 1                BC:= TLI[TLSC]
28HK0-> 11   2B   25 Z E 0                TLSC
 8HK1   12   2A   32767 X 0 B P           TLI[TLSC - 1] <> blokbeginmarker?
        13 N 2T    7 H K 3 A     ->       zo neen, dan blok uitluiden
        14   2A   26 Z E 2   Z            RHT = 0?
        15 N 7Y   22   C 0                zo neen, dan stop
        16   2A   27 Z E 2   Z            VHT = 0?
        17 N 7Y   23   C 0                zo neen, dan stop
        18 U 1S   91       A Z            DL = ;?
        19 Y 2T   15 H K 0 A     ->       zo ja, dan prescan vervolgen
        20   1B    1       A              verwijder begin uit TLI:
        21   6B   25 Z E 0                TLSC:= TLSC - 1
        22 U 1B    8 Z E 1   Z            TLSC = 0?
        23 N 2T   15 H K 0 A     ->       zo neen, dan prescan vervolgen
        24   2T    0 H L 0 A     =>       naar EPS, want prescan voltooid
0HK1 => 25   2A    1       A
        26 U 1S   98       A Z            DL = (?
        27 Y 4A   26 Z E 2                zo ja, dan RHT:= RHT + 1
        28 N 5A   26 Z E 2                zo neen, dan RHT:= RHT - 1
        29   2T   15 H K 0 A     =>       vervolg prescan
2HK1 => 30   2A    1       A
        31 U 1S   100      A Z            DL = [?
             DC D0
```

```
    DA   0 H K 4        DI
0 Y 4A  27 Z E 2                 zo ja, dan VHT:= VHT + 1
1 N 5A  27 Z E 2                 zo neen, dan VHT:= VHT - 1
2   2T  15 H K 0 A      =>       vervolg prescan
    DC D0
```

```
      EPS      end of prescan                                         HL0

           DA    0 H L 0       DI
24HK3=>  0   2A  12       A
         1   6A  26   X 0               klasse 6 in neutrale toestand
         2   0Y   0   XS                X1 horend
         3   6T  31 H R 0 0    =)       voorbereiding FOB6
         4   2A  15 Z E 2               NSS-vlag op
         5   7A  15 Z E 2               lezen uit magazijn zetten
         6   2A   0       A
         7   6A  14 Z E 2               RNS weer maagdelijk
         8   6A   1 Z E 0               IFLA:= 0
         9   6A   4 Z E 0               MFLA:= 0
        10   6A   6 Z E 0               VFLA:= 0
        11   6A   7 Z E 0               BN:= 0
        12   6A  13 Z E 0               AFLA:= 0
        13   6A  17 Z E 0               SFLA:= 0
        14   6A  18 Z E 0               EFLA:= 0
        15   6A  24 Z E 0               RLSC:= 0
        16   6A  26 Z E 0               FLSC:= 0
        17   6A  27 Z E 0               KLSC:= 0
        18   6A   4 Z E 2               VLAM:= 0
        19   2A  19 Z E 1
        20   0A   1       A
        21   6A  12 Z E 0               FLIB:= vulplaats + 1
        22   0A  16       A
        23   6A  28 Z E 0               KLIB:= FLIB + 16
        24   0A  16       A
        25   6A  31 Z E 0               NLIB:= KLIB + 16
        26   0A   9 Z E 2               NLSC0
        27 U 5A  21 Z E 0   P           NLIB + NLSC0 < PLIB?
        28 N 7Y  25   C 0               zo neen, stop: programma te lang
        29   2A   9 Z E 2
        30   6A  30 Z E 0               NLSC:= NLSC0
        31   2A   8 Z E 1               TLIB
           DC D0
```

HL1

```
          DA   0 H L 1       DI
       0  6A  25 Z E 0                TLSC:= 0
       1  2A   3 R K 0
       2  6A  26 Z E 1                GVC:= GVC0
       3  2S  161      A
       4  6T   0 Z T 0 0     =)       FTL met blokbeginmarker
       5  2A   9 Z E 2
       6  6A   0   X 0                aantal:= NLSC0
13 ->  7  2B   0   X 0                CYCLUS TRANSPORT PREVULLING NLI
       8  0B  17 Z E 2                PNLIB
       9  2S  32767 X 0 B             S:= PNLI[telling]
      10  2B   0   X 0
      11  0B  31 Z E 0                NLIB
      12  6S  32767 X 0 B             NLI[telling]:= S
      13  4T   7 H L 1 0 P   ->
      14  6T   6 H W 0 0     =)       INB
      15  6T   7 L L 1 0     =)       voorbereiding BSM
      16  2A  96       A              OPC van START
      17  6T   0 Z F 0 0     =)       FRL met START
      18  2T   0 E L 0 A     =>       naar basiscyclus
          DC D0
```

```
FOB6    fill output buffer class 6                                   HRO

aanroepen                        6T  0 H R0 0  =)  FOB6

                                 6T 31 H R0 0  =) voorbereiding FOB6


         DA   0 H R 0       DI
   =)  0    2A   17 Z E 1   P          typen onderdrukken?
       1 Y 2T    8   X 0    E   ->     zo ja, dan al klaar
       2   2B    1 Z E 2               vulplaats typmagazijn
  4 -> 3 U 1B    2 Z E 2    Z          magazijn vol?
       4 Y 1T    2      A       ->     zo ja, wacht dan
       5   0B    0 Z E 2               BOB6
       6   6S    0   X 0 B             berg symbool
       7   2A    1 Z E 2               vulplaats
       8   4P        AS
       9   0A    1      A              ophogen
      10   2LA 63       A              en wel cyclisch modulo 64
      11   6A    1 Z E 2               nieuwe vulplaats
      12   2A    2 Z E 2    P          typprogramma nog lopende?
      13 Y 2T    8   X 0    E   ->     zo ja, dan klaar
      14   6S    2 Z E 2               leegplaats:= oude vulplaats
      15   0Y   126  XS                standaardingang typprogramma
      16   2A    8   X 0
      17   6T    8   D 1 14     =)
 29 -> 18   2B    2 Z E 2               ledigplaats
      19   0B    0 Z E 2               BOB6
      20   2S    0   X 0 B             haal symbool
      21   6T   15   D 1 14     =)     TPWW
      22   6Z    2   XP                typ
      23   2A    2 Z E 2               ledigplaats
      24   0A    1      A              ophogen
      25   2LA 63       A              en wel cyclisch modulo 64
      26 U 1A    1 Z E 2    Z          magazijn leeg?
      27 Y 3A    1      A              zo ja, dan ledigplaats < 0 zetten
      28   6A    2 Z E 2               nieuwe ledigplaats
      29 N 2T   18 H R 0 A      ->     zo neen, dan typen voortzetten
      30   2T   13   D 1 A      =>     standaarduitgang typprogramma
   =) 31   2S    1      A              VOORBEREIDING
         DC D0
```

```
    DA    0 H R 1        DI
0   6S    1 Z E 2                 vulplaats:= 1
1   7S    2 Z E 2                 ledigplaats < 0: magazijn leeg
2   2A    3   D 0                 d1 van consolewoord
3   1P    2   AA                  in tekenbit schuiven
4   7A   17 Z E 1                 zet typvergunning
5   2T    8   X 0    E   =>       klaar
    DC D0
```

```
OCT      offer character to typewriter                               HS0

aanroep                          6T 0 H S0 1  =)  OCT



          DA    0 H S 0       DI
   =)  0   6A  13 Z E 1               red A
       1   6S  14 Z E 1               red S
       2   6B  15 Z E 1               red B
       3   2LA 63       A             isoleer karakter
       4 U 0LA 63       A Z           karakter = loos?
       5 Y 2T  23 H S 0 A     ->      zo ja, dan klaar
       6 U 1A  36       A P           karakter een hoofdletter?
       7 Y 2S  18   X 4 A             zo ja, dan S:= UC
       8 N 2S  19   X 4 A             zo neen, dan S:= LC
       9 U 1S  11 Z E 1   Z           klopt de shift?
      10 N 6S  11 Z E 1               zo neen, berg nieuwe shift en
      11 N 6T   0 H R 0 0     =)      FOB6 met shift
      12   2S  13 Z E 1               herleiding van code:
      13   2LS 63       A             isoleer karakter
      14 U 1S   9       A P           letter?
      15 N 0S   0   X 4 A             zo neen, dan + typbit
      16 N 2T  20 H S 0 A     ->      en klaar
      17 U 1S  35       A P           hoofdletter?
      18 Y 0S  48   X 2 A
      19 N 0S  75   X 2 A
16 -> 20   6T   0 H R 0 0     =)      FOB6 met karakter
      21   2B  15 Z E 1               herstel B
      22   2S  14 Z E 1               herstel S
 5 -> 23   2A  13 Z E 1               herstel A
      24   2T   9   X 0   E   =>      klaar
           DC D0
```

```
       NSS      next ALGOL symbol in S-register                              HT0



             DA   0 H T 0        DI
2ZY0 =>  0   3S  21 Z E 2   P          symbool in voorraad?
         1 Y 6S  21 Z E 2              zo ja, dan voorrad op leeg
24HT2    2 N 6T   0 L K 0 14    =)     RFS als geen voorraad
  14 ->  3 U 1S  101       A P         ingewikkeld?
         4 Y 2T   7 H T 0 A     ->     zo ja, dan uitzoeken
12,24->  5   2T   3 Z Y 0 A     =>     terug naar RNS
         6   2T   5 Z Y 0 A            (overbodig)
   4 =>  7 U 0LS 123       A Z         spatie?
         8 Y 2S  93        A           interne representatie voor spatie
         9 U 1S  119       A P         verschillend van spatie, tab, twnr?
        10 Y 2T  15 H T 0 A     ->     dan analyse voortzetten
        11   2A   9 Z E 1   Z          QC = 0? dwz., buiten string?
        12 N 2T   5 H T 0 A     ->     zo neen, dan niet skippen
        13   6T   0 L K 0 14    =)     RFS
        14   2T   3 H T 0 A     =>     nieuw symbool gaan onderzoeken
  10 => 15 U 1S  161       A P         is het | of _?
        16 Y 2T  25 H T 0 A     ->     dan samengesteld
        17 U 0LS 124       A Z         is het een :?
        18 N 7Y  14   C 0              zo neen, stop: ? of " of '
        19   6T   0 L K 0 14    =)     RFS voor symbool na :
        20 U 0LS 72        A Z         is het een =?
        21 N 7S  21 Z E 2              zo neen, dan in voorraad houden
        22 N 2S  90        A           en interne representatie voor :
        23 Y 2S  92        A           zo ja, interne representatie voor :=
        24   2T   5 H T 0 A     =>     en klaar
  16 => 25 U 0LS 162       A Z         is het |?
9HT1 -> 26   6T   0 L K 0 14    =)     RFS voor volgsymbool
        27 N 2T  11 H T 1 A     ->     zo neen, ga _ onderzoeken
        28 U 0LS 77        A Z         volgsymbool een ^?
        29 Y 2S  69        A           zo ja, dan interne representatie voor **
        30 Y 2T   5 H T 0 A     ->     gaan afleveren
        31 U 0LS 72        A Z         volgsymbool een =?
             DC D0
```

HT1

```
           DA   0 H T 1       DI
         0 Y 2S  75       A            zo ja, dan interne representatie voor <>
         1 Y 2T   5 H T 0 A     ->     gaan afleveren
         2 U 0LS 74       A Z          volgsymbool een <?
         3 Y 2S  102      A            zo ja, dan interne representatie voor (*
         4 Y 2T   5 H T 0 A     ->     gaan afleveren
         5 U 0LS 70       A Z          volgsymbool een >?
         6 Y 2S  103      A            zo ja, dan interne representatie voor *)
         7 Y 2T   5 H T 0 A     ->     gaan afleveren
         8 U 0LS 162      A Z          volgsymbool een |?
         9 Y 2T  26 H T 0 A     ->     zo ja, dan herhaling, dus skip
        10   7Y  11   C 0              en anders stop: ontoelaatbaar
27HT0=> 11 U 1S   9       A P          UNDERLINING
   29   12 U 1S  38       A E          verschillend van letter a t/m B?
        13 N 2T  30 H T 1 A     ->     zo neen, dan word delimiter
        14 U 1S  70       A Z          volgsymbool een >?
        15 Y 2S  71       A            zo ja, dan interne representatie voor >=
        16 Y 2T   5 H T 0 A     ->     gaan afleveren
        17 U 1S  76       A E          volgsymbool niet < of not of =?
        18 Y 2T  23 H T 1 A     ->     dan verder uitzoeken
        19 U 0LS 72       A Z          was het een =?
        20 Y 2S  80       A            zo ja, dan interne representatie voor eqv
        21 N 0LS  3       A            zo neen, dan die voor <= of imp
        22   2T   5 H T 0 A     =>     gaan afleveren
  18 => 23 U 0LS 124      A Z          volgsymbool een :?
        24 Y 2S  68       A            zo ja, dan interne representatie voor div
        25 Y 2T   5 H T 0 A     ->     gaan afleveren
        26 U 0LS 163      A Z          volgsymbool een _?
        27 N 7Y  13   C 0              zo neen, dan stop: ontoelaatbaar
        28   6T   0 L K 0 14    =)     RFS voor symbool na __
        29   2T  11 H T 1 A     =>     en onderzoek herhalen
  13 => 30   4P     SB                 OPBOUW WORD DELIMITER
        31   2S  13 H T 3 B            pak codewoord uit tabel
           DC D0
```

HT2

```
              DA   0 H T 2       DI
           0   2LS 127      A            isoleer waarde
           1 U 3LS 63       A Z          < 63? dwz., dubbelzinnig?
           2 Y 2T  14 H T 2 A     ->     zo ja, dan nader onderzoeken
       ->  3   6S  20 Z E 2              red gevonden interne representatie
    10 ->  4   6T   0 L K 0 14    =)     RFS     CYCLUS SKIP ONDERSTREEPTE
           5 U 0LS 163      A Z          een _?                     SYMBOLEN
           6 N 2T  11 H T 2 A     ->     zo neen, dan einde word delimiter
     9 ->  7   6T   0 L K 0 14    =)     RFS
           8 U 0LS 163      A Z          volgsymbool een herhaling van _?
           9 Y 2T   7 H T 2 A     ->     zo ja, dan skippen
          10   2T   4 H T 2 A     =>     volgend symbool gaan lezen
     6 => 11   7S  21 Z E 2              berg eerste niet-onderstreepte symbool
          12   2S  20 Z E 2              haal interne representatie weer op
          13   2T   5 H T 0 A     =>     en lever delimiter af
     2 => 14   4P        SS   Z          waarde = 0?
          15 Y 7Y  13   C 0              zo ja, dan ontoelaatbaar volgsymbool
          16 U 0LS  1       A Z          waarde = 1? dwz., een _c?
          17 N 2T  26 H T 2 A     ->     zo neen, dan tweede letter nodig
          18   2S   9 Z E 1 Z            QC = 0? dwz., buiten string?
          19 N 2S  97       A            zo neen, dan int. repr. voor comment-sym.
          20 N 2T   3 H T 2 A     ->     en delimiter aflezen en afleveren
          21   6T   0 L K 0 14    =)     RFS     SKIP COMMENTAAR
    25 -> 22   0LS 91       A Z          gelezen symbool al een ;?
          23   6T   0 L K 0 14    =)     RFS voor volgsymbool
          24 Y 2T   3 H T 0 A     ->     zo ja, dan opnieuw beginnen
          25   2T  22 H T 2 A     =>     anders skippen voortzetten
    17 => 26   6S  21 Z E 2              red eerste letter
          27   6T   0 L K 0 14    =)     RFS voor underlining
          28 U 0LS 163      A Z          gelezen symbool inderdaad een _?
          29 N 7Y  12   C 0              zo neen, stop: underlining ontbreekt
  OHT3 -> 30   6T   0 L K 0 14    =)     RFS voor tweede letter
          31 U 0LS 163      A Z          volgsymbool een herhaling van _?
              DC D0
```

HT3

```
             DA    0 H T 3       DI
           0 Y 2T   30 H T 2 A      ->       zo ja, dan skippen
           1 U 1S    9       A P
           2 U 1S   32       A E             verschillend van a t/m w?
           3 Y 7Y   13    C 0                zo ja, dan ontoelaatbaar
           4 U 0LS  29       A Z             letter een t?
           5 Y 2T   13 H T 3 A      ->       zo ja, dan derde letter nodig
           6   4P        SB
           7   2S   13 H T 3 B               pak codewoord uit tabel
           8   3P    7   SS    Z             isoleer waarde; dubbelzinnig?
           9 N 2T    3 H T 2 A      ->       zo neen, dan delimiter nu bekend
          10   2S   21 Z E 2                 pak anders de eerste letter weer op
          11   0LS  64       A               + 64, en nu wordt delimiter bekend
          12   2T    3 H T 2 A      =>       op grond van eerste letter
   5 =>   13   6T    0 L K 0 14     =)       RFS voor underlining
          14 U 0LS 163       A Z             gelezen symbool inderdaad een _?
          15 N 7Y   12    C 0                zo neen, stop: underlining ontbreekt
  18 ->   16   6T    0 L K 0 14     =)       RFS voor derde letter
          17 U 0LS 163       A Z             volgsymbool een herhaling van _?
          18 Y 2T   16 H T 3 A      ->       zo ja, dan skippen
          19 U 0LS  14       A Z             derde symbool een e?
          20 Y 2S   94       A               zo ja, dan int. repr. voor step
          21 N 2S  113       A               zo neen, dan die voor string
          22 Y 2T    3 H T 2 A      =>       en delimiter aflezen en afleveren
          23 DN     +15086                   a       117, 110       false   array
          24        +43                      b         0,  43
          25        +1                       c         0,   1               comment
          26        +86                      d         0,  86               do
          27        +13353                   e       104,  41       begin
          28        +10517                   f        82,  21       if
          29        +81                      g         0,  81               goto
          30        +10624                   h        83,   0       then
          31        +44                      i         0,  44
             DC D0
```

HT4

```
    DA   0 H T 4      DN
 0      +0                 j         0,   0
 1      +0                 k         0,   0
 2      +10866             l         0, 114        else    label
 3      +0                 m         0,   0
 4      +0                 n         0,   0
 5      +106               o         0, 106                own
 6      +112               p         0, 112                procedure
 7      +0                 q         0,   0
 8      +14957             r       116, 109        true    real
 9      +2                 s         0,   2
10      +2                 t         0,   2
11      +95                u         0,  95                until
12      +115               v         0, 115                value
13      +14304             w       111,  96        switch  while
14      +0                 x         0,   0
15      +0                 y         0,   0
16      +0                 z         0,   0
17      +0                 loos      0,   0
18      +0                 A         0,   0
19      +107               B         0, 107                Boolean
    DC D0
```

```
      INB      introduction new block                                      HW0

      aanroepen                    6T 0 H W0 0  =)  INB, BN:= BN + 1 inclusive
                                   6T 2 H W0 0  =)  INB without BN:= BN + 1
                                   6T 6 H W0 0  =)  INB without both BN:= BN + 1
                                                          and filling of TLI


           DA   0 H W 0       DI
   =)  0   2A   1       A
       1   4A   7 Z E 0             BN:= BN + 1
   =)  2   2A   8   X 0             red de link in het A-register
       3   2S  30 Z E 0             NLSC
       4   6T   0 Z T 0 0     =)    FTL met NLSC
13HW1  5   2T  10 H W 1 A     =>    ga TLI vullen met blokbeginmarker
 -> =) 6   2A  24 Z K 0
6HW1 -> 7  6A  21 Z E 1             INBA:= d17 + d15
       8   2B  21 Z E 0             B:= PLIB
       9   2S   0   X 0 B
      10   6S  21 Z E 0             PLIB:= PLI[0]
      11   0B   1       A           B:= B + 1
3HW1 -> 12 U 1B  21 Z E 0   Z       B = nieuwe PLIB?
      13 Y 2T   4 H W 1 A     ->    zo ja, dan groep afgehandeld
      14   2S   0   X 0 B           pak INW uit PLI
      15 U 2LS  7       A Z         enkelwoordsnaam?
      16 Y 0B   1       A           zo ja, dan B:= B + 1
      17 N 2A   1   X 0 B           zo neen, dan ook FNW pakken
      18 N 0B   2       A           en B:= B + 2
      19   6B  22 Z E 1             red B in INBB
      20   2B  30 Z E 0             NLSC
      21   0B  31 Z E 0             NLIB
      22 N 6A   0   X 0 B           zo neen, dan NLI[NLSC]:= FNW
      23 N 0B   1       A           en NLSC:= NLSC + 1
      24   6S   0   X 0 B           NLI[NLSC]:= INW
      25   0B   2       A           NLSC:= NLSC + 2
      26 U 1B  22 Z E 1   P         NLIB + NLSC > INBB
      27 Y 7Y  15   C 0             zo ja, dan stop: NLI groeit in PLI
      28   2A   7 Z E 0             BN voor constructie van ID
      29   2P  19   AA              * 2**19
      30   0A  21 Z E 1             + INBA
      31   6A  32767 X 0 B          NLI[NLSC - 1]:= ID
           DC D0
```

HW1

```
            DA   0 H W 1      DI
        0   1B  31 Z E 0              NLIB
        1   6B  30 Z E 0              vul nieuwe NLSC in
        2   2B  22 Z E 1              herstel B uit INBB
        3   2T  12 H W 0 A    =>      volgende naam overhevelen
13HW0=> 4   2A  23 Z K 0
        5 U 1A  21 Z E 1   Z          INBA = d18 + d15?
        6 N 2T   7 H W 0 A    ->      zo neen, dan INBA:= d18 + d15 en
        7   2A   0       A                           volgend stuk doen
        8   6A  25 Z E 1              LVC:= 0
        9   2T   8   X 0   E  =>      klaar
5HW0 -> 10  2S  161      A
       11   6T   0 Z T 0 0    =)      FRL met blokbeginmarker
       12   6A   8   X 0              herstel link uit A
       13   2T   6 H W 0 A    =>      ga namen uit PLI overhevelen
            DC D0
```

```
NBD      new block as result of declaration?                       HU0

aanroep                          6T 0 H U0 1  =)  NBD



            DA   0 H U 0       DI
      =)  0   2B  25 Z E 0            TLSC
          1   2S  32766 X 0 B         TLI[TLSC - 2]
          2   0LS 161      A Z        blokbeginmarker onder top van TLI?
          3 Y 2T   9   X 0   E   ->   zo ja, dan klaar: geen nieuw blok
          4   1B   1       A          verwijder begin uit TLI:
          5   6B  25 Z E 0            TLSC:= TLSC - 1
          6   2S   6 Z K 0
          7   2A   0       A
          8   6T   0 Z F 0 0    =)    FRL met 2A  0 A
          9   2S   1 Z K 0
         10   2A   1       A          opc1: relatief tov RLIB
         11   0S  24 Z E 0            RLSC
         12   0S   3       A          + 3 geeft beginadres anonym blok
         13   6T   0 Z F 0 0    =)    FRL met X1X 2B 'RLSC + 3' A
         14   2S   0       A
         15   2A   9       A          OPC van ETMP
         16   6T   0 Z F 0 0    =)    FRL met ETMP
         17   2S   8 Z K 0
         18   0S  26 Z E 0            FLSC
         19   2A   2       A          opc2: referentie naar FLI
         20   6T   0 Z F 0 0    =)    FRL met X2X 2T 'FLSC'
         21   2S  26 Z E 0            FLSC
         22   6T   0 Z T 0 0    =)    FTL met FLSC
         23   2S   1       A
         24   4S  26 Z E 0            FLSC:= FLSC + 1
         25   6T   0 H W 0 0    =)    INB
         26   2S  104      A
         27   6T   0 Z T 0 0    =)    FRL met begin
18HY0=)  28   2S   1 Z K 0
         29   0S   7 Z E 0            BN
         30   2A   0       A
         31   6T   0 Z F 0 0    =)    FRL met 2B 'BN' A
              DC D0
```

HU1

```
    DA   0 H U 1        DI
0   2S   0       A
1   2A  89       A                OPC van SCC
2   6T   0 Z F 0 0      =)        FRL met SCC
3   2A   7 Z E 0                  BN
4   0A  160      A                + 5 * 32
5   6A  23 Z E 1                  PNLV
6   6A   4 Z E 2                  maak VLAM <> 0
7   2T   9   X 0   E    =>        klaar
    DC D0
```

```
DDEL  procedure                                                      HY0


           DA    0 H Y 0        DI
   =>  0   2T   14 H Y 0 A      =>        doe eerst RLA en NBD?
16 =>  1   2S    8 Z K 0
       2   0S   26 Z E 0                  FLSC
       3   2A    2       A                opc2: referentie naar FLI
       4   6T    0 Z F 0 0      =)        FRL met X2X 2T 'FLSC'
       5   2S   26 Z E 0                  FLSC
       6   6T    0 Z T 0 0      =)        FTL met FLSC
       7   2S    1       A
       8   4S   26 Z E 0                  FLSC:= FLSC + 1
       9   6T    0 F T 0 2      =)        RND voor procedure identifier
      10   6T    0 H Z 0 0      =)        LFN
      11   6T    0 F Y 0 2      =)        LDEC
      12   6T    0 H W 0 0      =)        INB
      13   2T   18 H Y 0 A      =>
 0 => 14   6T    0 K N 0 2      =)        RLA
      15   6T    0 H U 0 1      =)        NBD?
      16   2T    1 H Y 0 A      =>        ga sprong over body produceren
      17   4S    7 Z E 0                  (overbodig)
13 => 18   6T   28 H U 0 1      =)        NBD-gedeeltelijk
      19   2A    9 Z E 0                  DL
      20   0LA  91       A Z              = ;?
      21 Y 2T    0 E L 0 A      ->        zo ja, dan terug naar basiscyclus
31 -> 22   6T    0 F T 0 2      =)        RND voor formele parameter
      23   2A   23 Z E 1                  PNLV voor constructie ID
      24   0A   27 Z K 0                  d16 + d15: indicatie formeel en dynamisch
      25   6A    8 Z E 0                  ID voorlopig voltooid
      26   6T    0 H N 0 0      =)        FNL
      27   2A   64       A                2 * 32
      28   4A   23 Z E 1                  PNLV:= PNLV + 64 als PARD-reservering
      29   2A    9 Z E 0                  DL
      30   0LA  87       A Z              = ,?
      31 Y 2T   22 H Y 0 A      ->        zo ja, dan volgende formele parameter
           DC D0
```

```
             DA   0 H Y 1       DI
        0    6T   0 F T 0 2     =)       RND voor ; na )
10HY3-> 1    6T   0 F T 0 2     =)       RND
  20    2    2A  29 Z E 0   Z            NFLA = 0? dwz., kale delimiter?
        3 N 2T    1 E L 0 A     ->       zo neen, dan terug naar basiscyclus
        4    2A   9 Z E 0                DL
        5 U 0LA 104       A Z            = begin?
        6 Y 2T    0 E H 0 A     ->       zo ja, dan naar DDEL
        7 U 0LA 115       A Z            DL = value?
        8 N 2T   21 H Y 1 A     ->       zo neen, dan specificaties
23,26   9    2S  28 Z K 0                d26 als valuevlag
28,31-> 10   6S   5 Z E 2                zet SPE
13HY2-> 11   6T   0 F T 0 2     =)       RND voor identifier uit list
  19    12   6T   0 H Z 0 0     =)       LFN
        13   2B  22 Z E 0                NID
        14   0B  31 Z E 0                NLIB
        15   2S   5 Z E 2                SPE toevoegen
        16   4S   0   X 0 B              aan ID in NLI
        17   2A   9 Z E 0                DL
        18   0LA 87       A Z            = ,?
        19 Y 2T  11 H Y 1 A     ->       dan volgende identifier uit list
        20   2T   1 H Y 1 A     =>       ga testen op begin van body
   8 => 21 U 0LA 113      A Z            DL = string?
1HY2 -> 22 Y 2S   0       A              zo ja, dan lege SPE-vlag
        23 Y 2T  10 H Y 1 A     ->       en ga specification part lezen
        24 U 0LA 114      A Z            DL = label?
        25   2S  12 Z K 0                zo ja, neem d17 als SPE-vlag
        26 Y 2T  10 H Y 1 A     ->       en ga specification part lezen
        27 U 0LA 111      A Z            DL = switch?
        28 Y 2T  10 H Y 1 A     ->       zo ja, ga sp. part lezen met d17 als SPE
        29 U 0LA 112      A Z            DL = procedure?
11HY2-> 30 Y 2S  29 Z K 0                zo ja, neem d18 als SPE-vlag
        31 Y 2T  10 H Y 1 A     ->       en ga specification part lezen
             DC D0
```

HY2

```
           DA   0 H Y 2      DI
         0 U 0LA 110      A Z          DL = array?
         1 Y 2T  22 H Y 1 A     ->     zo ja, ga sp. part lezen met lege
         2 U 1A  109      A Z          DL = real?              SPE-vlag
         3 Y 2S   0       A            zo ja, dan d19 = 0 nemen
         4 N 2S   4 Z K 0              zo neen, dan d19 = 1 als integerbit
         5 U 1A  106      A E          DL geen specificator?
         6 Y 2T   0 E H 0 A     ->     zo ja, naar DDEL wegens if, for, goto
         7   6S   5 Z E 2              zet SPE
         8   6T   0 F T 0 2     =)     RND voor delimiter na real, integer,
         9   2A   9 Z E 0              DL                        of Boolean
        10 U 0LA 112      A Z          = procedure?
        11 Y 2T  30 H Y 1 A     ->     zo ja, dan net als non-type procedure
        12 U 0LA 110      A Z          DL = array?
        13 Y 2T  11 H Y 1 A     ->     zo ja, ga specification part lezen
9HY3 -> 14   6T   0 H Z 0 0     =)     LFN, want blijkbaar identifier gelezen
        15   2B  22 Z E 0              NID
        16   0B  31 Z E 0              NLIB
        17   2S   5 Z E 2              SPE toevoegen
        18   4S   0   X 0 B P          aan ID toevoegen; called by name?
        19 Y 2T   6 H Y 3 A     ->     zo ja, dan eenvoudig
        20   3S   3 Z K 0              d16       PRODUCTIE VALUE PROGRAMMA
        21   1S  28 Z K 0              d26
        22   2LS  0   X 0 B            schrappen uit
        23   6S   0   X 0 B            ID in NLI
        24   6S   8 Z E 0              wijzig ID conform
        25   6T   0 Z R 0 0     =)     AVR voor 2S 'pardpositie' A
        26   2A   5 Z E 2   Z          SPE = 0? dwz., real?
        27 Y 2A  14       A            zo ja, dan OPC van TRAD
        28 N 2A  16       A            zo neen, dan OPC van TIAD
        29   2S  0        A
        30   6T   0 Z F 0 0     =)     FRL met TRAD of TIAD
        31   6T   0 Z R 0 0     =)     AVR voor 2S 'pardpositie' A
           DC D0
```

HY3

```
            DA   0 H Y 3        DI
        0   2S   0        A
        1   2A  35        A                 OPC van TFR
        2   6T   0 Z F 0 0      =)          FRL met TFR
        3   2S   0        A
        4   2A  85        A                 OPC van ST
        5   6T   0 Z F 0 0      =)          FRL met ST
19HY2-> 6   2A   9 Z E 0
        7   0LA 87        A Z               DL = ,?
        8 Y 6T   0 F T 0 2      =)          zo ja, dan RND voor behandeling
        9 Y 2T  14 H Y 2 A      ->          volgende identifier uit lijst
       10   2T   1 H Y 1 A      =>          ga testen op begin van body
            DC D0
```

```
   FNL      fill name list                                               HNO

   aanroep                          6T 0 H NO 0  =)  FNL



        DA   0 H N 0       DI
=)  0   2B  30 Z E 0                NLSC
    1   0B  31 Z E 0                NLIB
    2   0B   2 Z E 1                DFLA
    3   0B   2       A              NLSC:= NLSC + DFLA + 2
    4 U 1B  21 Z E 0   P            NLSC + NLIB > PLIB?
    5 Y 7Y  16   C 0                zo ja, stop: NLI groeit in PLI
    6   2A   8 Z E 0
    7   6A  32767 X 0 B             NLI[NLSC - 1]:= ID
    8   2A   1 Z E 1
    9   6A  32766 X 0 B             NLI[NLSC - 2]:= INW
   10 U 2LA  7       A Z            enkelwoordsnaam?
   11 N 2A   0 Z E 1                zo neen, dan
   12 N 6A  32765 X 0 B             NLI[NLSC - 3]:= FNW
   13   1B  31 Z E 0                NLIB
   14   6B  30 Z E 0                vul nieuwe NLSC in
   15   2T   8   X 0   E   =>       klaar
        DC D0
```

```
DDEL  integer                                                    KZ0
DDEL  Boolean


               DA   0 K Z 0        DI
       =>  0   2A   1       A
           1   6A  24 Z E 1                  IBD:= 1
           2   6T   0 H U 0 1      =)        NBD?
           3   6T   0 F T 0 2      =)        RND
           4   2A  29 Z E 0   Z              NFLA = 0? dwz., geen identifier?
           5 Y 2T  21 K Z 1 A      ->        zo ja, dan geen scalair
  5KE0 ->  6   2A   7 Z E 0   Z              BN = 0?
           7 Y 2T   9 K Z 1 A      ->        dan statische adressering verzorgen
    22 ->  8   2A  23 Z E 1                  PNLV voor constructie ID
  3KZ1     9   0A   0 Z K 0                  d15 als indicatie dynamisch adres
          10   2S  24 Z E 1   Z              IBD = 0?
          11 N 0A   4 Z K 0                  anders d19 als integerbit toevoegen
          12   6A   8 Z E 0                  ID klaar
          13 Y 2A  64       A                2 * 32 of
          14 N 2A  32       A                1 * 32
          15   4A  23 Z E 1                  ter ophoging van PNLV
          16   3P   5   AA                   2 of 1
          17   4A  25 Z E 1                  ter ophoging van LVC
          18   6T   0 H N 0 0      =)        FNL
          19   2A   9 Z E 0                  DL
          20 U 0LA 87       A Z              = ,?
          21 Y 6T   0 F T 0 2      =)        dan RND voor volgende identifier
          22 Y 2T   8 K Z 0 A      ->        en ID gaan construeren
          23   6T   0 F T 0 2      =)        RND voor delimiter na ;
          24   2A   9 Z E 0                  DL
          25 U 1A  109      A Z              = real?
          26 Y 2S   0       A
          27 N 2S   1       A                0 of 1
          28 U 1A  106      A E              DL <> real of integer of Boolean?
          29 N 2T   0 K Z 1 A      ->        zo neen, dan voortgezette declaratie
          30   6T   0 K Y 0 1      =)        RLV (want klaar met scalairen)
          31   2T   1 E L 0 A      =>        terug naar basiscyclus (zonder RND)
               DC D0
```

KZ1

```
              DA    0 K Z 1        DI
29KZ0=>  0    6S   24 Z E 1                  IBD:= 0 of 1
         1    6T    0 F T 0 2      =)        RND
         2    2A   29 Z E 0   Z              NFLA = 0? dwz., geen identifier?
         3 N  2T    8 K Z 0 A      ->        zo neen, dan declaratie van scalair
         4    6T    0 K Y 0 1      =)        RLV (want geen scalairen verder)
         5    2A    9 Z E 0                  DL
         6    0LA 110       A Z              = array?
         7 Y  2T    3 K F 0 A      ->        dan door naar DDEL array
7KZ0     8    2T    0 E H 0 A      =>        naar DDEL
  19 =>  9    2A   26 Z E 1                  GVC voor constructie van ID
9KH0    10    3S   24 Z E 1   Z              IBD = 0?
        11 N  0A    4 Z K 0                  anders d19 als integerbit toevoegen
        12    6A    8 Z E 0                  ID klaar
        13    0S    2       A                2 of 1
        14    4S   26 Z E 1                  ter ophoging van GVC
        15    6T    0 H N 0 0      =)        FNL
        16    2A    9 Z E 0                  DL
        17 U  0LA  87       A Z              = ,?
        18 Y  6T    0 F T 0 2      =)        zo ja, dan RND voor volgende identifier
        19 Y  2T    9 K Z 1 A      ->        en ID gaan construeren
        20    2T    0 E L 0 A      =>        terug naar basiscyclus
5KZ0 => 21    2A    9 Z E 0                  DL
        22    0LA 110       A Z              = array?
        23 N  2T    0 H Y 0 A      ->        zo neen, dan naar DDEL procedure
        24    2T    3 K F 0 A      =>        door naar DDEL array
              DC D0
```

```
  DDEL  real                                                        KE0



      DA   0 K E 0       DI
=>  0   2A   0       A
    1   6A  24 Z E 1                IBD:= 0
    2   6T   0 H U 0 1     =)       NBD?
    3   6T   0 F T 0 2     =)       RND
    4   2A  29 Z E 0   Z            NFLA = 0? dwz., geen identifier?
    5 N 2T   6 K Z 0 A     ->       zo neen, dan verder samen met DDEL integer
    6   2T   0 E H 0 A     =>       naar DDEL
        DC D0
```

```
     DDEL  array                                                   KF0

              DA    0 K F 0       DI
     =>  0    2A    0       A
         1    6A   24 Z E 1                 IBD:= 0
 7KZ1    2    6T    0 H U 0 1     =)        NBD?
24KZ1->  3    2A    7 Z E 0   Z             BN = 0?
         4 N  2T   20 K F 2 A     ->        zo neen, dan dynamische grenzen
8KF0->   5    2A   14 Z K 0                 d25, d24 als indicatie KLI
         6 U  2A   24 Z E 1   Z             IBD = 0? dwz., real array?
         7 N  0A    4 Z K 0                 anders d19 als intergerbit toevoegen
         8    6A    8 Z E 0                 voorlopige ID
18KF2->  9    2A   30 Z E 0                 NLSC
        10    6A   23 Z E 0                 dumpen in ARRA
        11    2A   25 Z E 0                 TLSC
        12    6A   29 Z E 1                 dumpen in ARRB
  17 -> 13    6T    0 F T 0 2     =)        RND voor array identifier
        14    6T    0 H N 0 0     =)        FNL
        15    2A    9 Z E 0                 DL
        16    0LA 100       A Z             = [?
        17 N  2T   13 K F 0 A     ->        anders volgende identifier lezen
        18    6A   30 Z E 1                 ARRC:= 0
        19    2S    2       A               CONSTRUCTIE STOFU
        20    1S   24 Z E 1                 2 - IBD
        21    6T    0 Z T 0 0     =)        FTL met delta[0]
18KF1-> 22    6T    0 F T 0 2     =)        RND voor lower bound
        23    2A    9 Z E 0                 DL
        24 U  0LA  90       A Z             = :?
        25 Y  2T   28 K F 0 A     ->        dan klaar met lower bound
        26 U  0LA  64       A Z             DL = +?
        27    6T    0 F T 0 2     =)        RND voor unsigned number
  25 -> 28 Y  2S    1 Z E 1                 pak INW, dwz. L[i] met
        29 N  3S    1 Z E 1                 het juiste teken
        30    6S   31 Z E 1                 lower bound dumpen in ARRD
        31    6T    0 F T 0 2     =)        RND voor upper bound
              DC D0
```

KF1

```
            DA   0 K F 1       DI
        0   2A  29 Z E 0   Z             NFLA = 0? dwz., geen number?
        1 N 2T   5 K F 1 A     ->        anders klaar met upper bound
        2   2A   9 Z E 0                 DL
        3   0LA 65       A Z             = -?
        4   6T   0 F T 0 2     =)        RND voor unsigned number
   1 -> 5   2B  25 Z E 0                 TLSC
        6   2S  31 Z E 1                 pak in ARRD gedumpte lower bound
        7   3X  32767 X 0 B              * (-TLI[TLSC - 1])
        8   4S  30 Z E 1                 ARRC:= ARRC - L[i] * delta[i]
        9   3S  31 Z E 1                 pak - L[i]
       10 N 0S   1 Z E 1                 tel bij INW, dwz. U[i] met
       11 Y 1S   1 Z E 1                 het juiste teken
       12   0S   1       A
       13   2A   9 Z E 0                 DL
       14   0LA 101      A Z             = ]?
       15 N 2X  32767 X 0 B              delta[i+1]:= (U[i] - L[i] + 1) *
       16 Y 3X  32767 X 0 B                             delta[i]
       17   6T   0 Z T 0 0     =)        FTL met delta[i+1] of - delta[n]
       18 N 2T  22 K F 0 A     ->        zo nodig volgend bound pair lezen
       19   2B  30 Z E 0                 NLSC
14KF2-> 20   6B  31 Z E 1                 dumpen in ARRD
       21   0B  31 Z E 0                 NLIB
       22   2A  27 Z E 0                 KLSC als adres STOFU
       23   4A  32767 X 0 B              voltooi ID in NLI
       24   2S  26 Z E 1                 GVC als beginadres arraysegment
       25   6T   0 K U 0 0     =)        FKL met GVC
       26   2S  26 Z E 1                 GVC samen met
       27   0S  30 Z E 1                 ARRC het geextrapoleerde nulpunt
       28   6T   0 K U 0 0     =)        FKL met GVC + ARRC
       29   2B  29 Z E 1                 pak in ARRB gedumpte TLSC
5KF2 -> 30   2S   0   X 0 B P            TLI[TLSC] > 0?
       31 Y 0B   1       A               dan nog niet - delta[n]
            DC D0
```

KF2

```
            DA    0 K F 2       DI
         0 N 2B   29 Z E 1               anders de in ARRB gedumpte TLSC
         1   6B   25 Z E 0               TLSC resetten
         2 N 5S   26 Z E 1               en GVC ophogen met delta[n]
         3   6T    0 K U 0 0    =)       FKL met delta[i] of - delta[n]
         4 Y 2B   25 Z E 0               zo nodig TLSC pakken
         5 Y 2T   30 K F 1 A    ->       en volgende delta
         6   2B   31 Z E 1               pak de in ARRD gedumpte NLSC
         7   0B   31 Z E 0               NLIB
         8   2S   32766 X 0 B            pak NLI[NLSC - 2], dwz., INW uit NLI
         9   1B   31 Z E 0               NLIB
        10   2LS   7       A Z           enkelwoordsnaam?
        11 Y 1B    2       A             NLSC passend aflagen
        12 N 1B    3       A
        13 U 1B   23 Z E 0   Z           de in ARRA gedumpte NLSC al bereikt?
        14 N 2T   20 K F 1 A    ->       anders volgende STOFU gaan bouwen
        15   6T    0 F T 0 2    =)       RND voor delimiter na ]
        16   2A    9 Z E 0               DL
        17   0LA 87        A Z           = ,?
        18 Y 2T    9 K F 0 A    ->       dan voortgezette array-declaratie
        19   2T    0 F S 0 A    =>       door naar DDEL ;
4KF0 => 20   2A    0       A             DYNAMISCH ARRAY   ga vlaggen zetten:
29EF1   21   6A    3 Z E 2               IC:= 0
        22   6A    6 Z E 2               AIC:= 0
        23   6A    8 Z E 0               ID:= 0
  30 -> 24   2A    1       A
        25   4A    6 Z E 2               AIC:= AIC + 1
        26   6T    0 F T 0 2    =)       RND voor array identifier
        27   6T    0 H N 0 0    =)       FNL
        28   2A    9 Z E 0               DL
        29   0LA 87        A Z           = ,?
        30 Y 2T   24 K F 2 A    ->       dan tellen en volgende identifier
        31   2A    1       A
            DC D0
```

KF3

```
    DA   0 K F 3        DI
0   6A  18 Z E 0                EFLA:= 1
1   6A   0 Z E 0                OFLA:= 1
2   2T  14 F E 0 A       =>     OH:= 0; FTD; door naar basiscyclus
    DC D0
```

```
   DDEL  own                                                            KH0


        DA   0 K H 0       DI
=>  0   6T   0 H U 0 1     =)       NBD?
    1   6T   0 Z Y 0 0     =)       RNS voor delimiter na own
    2   2A   9 Z E 0                DL
    3   0LA 109      A Z            = real?
    4 N 2A   1       A              anders integer
    5   6A  24 Z E 1                IBD:= 0 of 1
    6   6T   0 F T 0 2     =)       RND
    7   2A  29 Z E 0   Z            NFLA = 0? dwz., geen identifier?
    8 Y 2T   5 K F 0 A     ->       dan als array van blok 0 behandelen
    9   2T   9 K Z 1 A     =>       anders als <type> van blok 0
        DC D0
```

```
DDEL  < <= = >= > <>                                                      KKO



                DA   O K K O       DI
      =>  0    2A   8       A                8 als OH
4KLO ->  1    2S   1       A
          2    6S   0 Z E 0                  OFLA:= 1
          3    2T   1 E T 0 A     =>        verder samen met DDEL * / div
                DC DO
```

```
   DDEL  not and or implies eqv                                        KL0


          DA   0 K L 0       DI
 =>  0 U 1B   76      A Z            DL = not?
     1   2A  83       A
     2   1A   9 Z E 0                construeer OH uit DL
     3 Y 2T   7 E Y 0 A     ->       dan FTD; OFLA:= 1; naar basiscyclus
     4   2T   1 K K 0 A     =>       anders samen met  DDEL < <= = >= > <>
         DC D0
```

```
  DDEL  goto                                                          KR0


        DA   0 K R 0        DI
=>  0   6T   0 K N 0 2      =)       RLA
    1   2T   0 E L 0 A      =>       terug naar basiscyclus
        DC D0
```

```
      DDEL  (*                                                        KS0

             DA   0 K S 0      DI
      =>  0  2A   1       A
          1  6A   9 Z E 1              QC:= 1
   22 ->  2  2B   0       A
          3  6B  28 Z E 1              QB:= 0
   18 ->  4  6B  27 Z E 1              QA:= (initieel) 0
          5  6T   0 Z Y 0 0    =)      RNS voor string-symbool
          6  2A   9 Z E 0              DL
          7  2S   1       A
          8 U 0LA 102     A Z          = (*?
          9 Y 4S   9 Z E 1             zo ja, dan QC:= QC + 1
         10 U 0LA 103     A Z          DL = *)?
         11 Y 5S   9 Z E 1   Z         zo ja, dan QC:= QC - 1; QC = 0?
         12  2B  27 Z E 1              QA als schuifwijzer
         13 Y 2T  23 K S 0 A   ->      dan einde string
         14  2P   0   AA  B            schuif symbool in goede positie
         15  4A  28 Z E 1              en tel bij woord-in-opbouw
         16  0B   8       A            schuifwijzer ophogen
         17 U 1B  24      A Z          maar
         18 N 2T   4 K S 0 A   ->      modulo 24
         19  2S  28 Z E 1              pak voltooid woord
         20  2A   0       A
         21  6T   0 Z F 0 0    =)      FRL met string-woord
         22  2T   2 K S 0 A    =>      start nieuw string-woord
   13 => 23  2S  255      A            eindmarker
         24  2P   0   SS  B            in goede positie schuiven
         25  0S  28 Z E 1              woord-in-opbouw erbij
         26  2A   0       A
         27  6A   0 Z E 0              OFLA:= 0
         28  6T   0 Z F 0 0    =)      FRL met laatste string-woord
         29  2T   0 E L 0 A    =>      terug naar basiscyclus
             DC D0
```

```
  DDEL  **                                                                KTO


        DA   0 K T 0      DI
=>  0   2A  11      A              11 als OH
    1   2T   1 E T 0 A    =>       verder samen met DDEL * / div
        DC D0
```

```
      END                                                              KW0

               DA    0 K W 0       DI
29FS1=>  0     2A   97         A             OPC van STOP
         1     6T    0 Z F 0 0     =)        FRL met STOP
         2     2S   24 Z E 0
         3     6S    3 R E 0                 RLSCE:= RLSC
         4     2S   27 Z E 0
         5     6S    4 R E 0                 KLSCE:= KLSC
         6     2S   26 Z E 1
         7     6S   21   X 1                 GVC naar goede plaats
         8     2S   26 Z E 0
         9     6S    0   X 0                 zet telling met FLSC
        10     2S   19 Z E 1
        11     6S   13 R E 0                 ledigplaats RBS:= vulplaats BSM
        12     2A    2 R K 0                 KLIE ter berekening van RLIB
        13     1A    3 R E 0                 RLSCE
        14     1A    4 R E 0                 KLSCE
        15     3LA 31          A             naar beneden afronden
        16     6A    1 R E 0                 RLIB klaar
        17     6A   11 R E 0
        18     2B   12 Z E 0
        19     6B    5 R E 0                 FLIB naar goede plaats
  22 -> 20     4A    0   X 0 B               cyclus voor
        21     0B    1         A             FLI[i]:= FLI[i] + RLIB
        22     4T   20 K W 0 0 P   ->
        23     6B    6 R E 0                 red FLIB + FLSCE
        24     6A   15   X 1                 MCPE:= RLIB (als startwaarde)
        25     0A    3 R E 0
        26     6A    2 R E 0                 KLIB:= RLIB + RLSCE
        27     2S   128        A
        28     6S    0   X 0                 lengte MLI:= 128
        29     2S    0         A
        30     2B    4 R K 0                 MLIB
1KW1 -> 31     6S    0   X 0 B               cyclus clear MLI:
               DC D0
```

```
             DA   0 K W 1       DI
          0  0B   1       A              MLI[MCPnr]:= +0
          1  4T  31 K W 0 0 P   ->
          2  6S  14 R E 0                aantal MCP's:= 0
          3  2B   9 Z E 2                PRIMAIRE MARKERING VAN MCP'S
          4  0B  31 Z E 0                NLIB + NLSC0
          5  2A  25 Z E 2                NLSCop
          6  0A  31 Z E 0                NLIB
          7  6A  22 Z E 0
          8  2T  29 K W 1 A     =>
   30 => 9   2S  32767 X 0 B             CYCLUS TEST PRIMAIR GEBRUIK
         10  2A  32766 X 0 B             pak ID en INW van MCP-naam uit NLI
         11  2LA  7       A Z            enkelwoordsnaam?
         12 Y 1B   2      A              NLSC passend aflagen
         13 N 1B   3      A
         14 U 2LS  0 Z K 0   Z           d15 van ID = 0?
         15 N 2T  29 K W 1 A    ->       zo neen, dan MCP ongebruikt
         16  6B  24 Z E 0                red NLIB + NLSC
         17  2A   1       A
         18  4A  14 R E 0                tel gebruikte MCP
         19  2LS 32767    A              isoleer FLSC uit ID
         20  0S  12 Z E 0                FLIB
         21  4P       SB
         22  2A   0   X 0 B              FLI[FLSC], bevat MCPnr+RLIB+d15+d18
         23  1A   1 R E 0                RLIB
         24  2LA 32767    A              isoleer MCPnr
         25  4P       AB
         26  0B   4 R K 0                MLIB
         27  7S   0   X 0 B              MLI[MCPnr]:= - (FLIB + FLSC)
         28  2B  24 Z E 0                herstel NLIB + NLSC
 8,15 -> 29 U 1B  22 Z E 0   P           > NLIB + NLSCop?
         30 Y 2T   9 K W 1 A    ->       dan volgende MCP onderzoeken
         31  2S   5 R K 0                CRFB
             DC D0
```

KW2

```
           DA   0 K W 2       DI
        0  0P   1   SS                 SECUNDAIRE MARKERING VAN MCP'S
        1  6S   0 R E 0                voorbereiding van HSC
16,30-> 2  6T   0 R S 0 0     =)       HSC voor lengte MCP; eindmarker?
        3 Y 2T 31 K W 2 A     ->       zo ja, dan klaar met CRF
        4  6S   7 R E 0                red lengte MCP
        5  6T   0 R S 0 0     =)       HSC voor nummer MCP
        6  6S   8 R E 0                red nummer MCP
        7  2A   0       A
        8  6A   9 R E 0                USE:= false
  14 -> 9  4P       SB
       10  0B   4 R K 0                MLIB
       11  2A   0   X 0 B Z            MLI[MCPnr] = 0? dwz., geen behoefte?
       12 N 6A  9 R E 0                anders USE:= true
       13  6T   0 R S 0 0     =)       HSC voor nummer MCP; eindmarker?
       14 N 2T  9 K W 2 A     ->       anders verder testen
       15  2A   9 R E 0   Z            USE = false?
       16 Y 2T  2 K W 2 A     ->       zo ja, dan volgende MCP onderzoeken
       17  2A  15   X 1                MCPE
       18  1A   7 R E 0
       19  6A  15   X 1                MCPE:= MCPE - lengte MCP
       20 U 1A  1 R K 0   P            MCPE > MCPB?
       21 N 7Y 25   C 0                anders stop: MCP zou in MLI zakken
       22  2B   8 R E 0                pak gered MCPnr
       23  0B   4 R K 0                MLIB
       24  3S   0   X 0 B P            MLI[MCPnr] < 0? dwz., primaire behoefte?
       25  6A   0   X 0 B              MLI[MCPnr]:= MCPE als beginadres MCP
       26 Y 4P      SB                 bij primaire behoefte ook:
       27 Y 6A  0   X 0 B              FLI[FLSC]:= MCPE
       28 N 2S  1       A              bij uitsluitend secundaire behoefte:
       29 N 4S 14 R E 0                tel gebruikte MCP
       30  2T   2 K W 2 A     =>       volgende MCP gaan onderzoeken
   3 => 31  2S   4 R E 0                KLSCE
           DC D0
```

KW3

```
            DA   0 K W 3       DI
       0    6S   0   X 0                zet telling met KLSCE
       1    2T   8 K W 3 A     =>
8 =>   2    2B  28 Z E 0                CYCLUS TRANSPORT KLI
       3    0B   0   X 0
       4    2S   0   X 0 B              pak KLI[KLSC]
       5    2B   2 R E 0
       6    0B   0   X 0
       7    6S   0   X 0 B              berg KLI[KLSC]
1 ->   8    4T   2 K W 3 0 E   ->
       9    2T   0 R Z 0 A     =>       naar naschouw-programma
            DC D0
```

```
FKL      fill constant list                                          KU0

aanroep                          6T 0 K U0 0  =)  FKL met S


          DA   0 K U 0       DI
21-> =)  0   2B  27 Z E 0              KLSC
         1   0B  28 Z E 0              KLIB
         2 U 1B  31 Z E 0   Z          KLIB + KLSC = NLIB?
         3 N 6S   0   X 0 B            zo neen, dan KLI[KLSC]:= S,
         4 N 2A   1       A
         5 N 4A  27 Z E 0              en KLSC:= KLSC + 1
         6 N 2T   8   X 0   E   =>     en klaar
         7   2B  30 Z E 0              OPSCHUIVEN VAN NLI
         8   6B   0   X 0              aantal:= NLSC
         9   0B  31 Z E 0              NLIB
        10   5P      BA
        11   1A  16       A
        12   0A  21 Z E 0   P          NLIB + NLSC + 16 < PLIB?
        13 N 7Y  18   C 0              zo neen, stop: NLI schuift in PLI
        14   2T  18 K U 0 A     =>
  18 => 15   1B   1       A            opschuifcyclus:
        16   2A   0   X 0 B            16 plaatsen
        17   6A  16   X 0 B            omhoog
  14 -> 18   4T  15 K U 0 0 E   ->
        19   2A  16       A
        20   4A  31 Z E 0              NLIB:= NLIB + 16
        21   2T   0 K U 0 A     =>     klaar met schuiven
          DC D0
```

```
RLV      reservation local variables                               KY0

aanroep                         6T 0 K Y0 1  =)  RLV


        DA   0 K Y 0       DI
=)  0   2S  25 Z E 1   Z           LVC = 0?
    1 Y 2T   9   X 0   E   ->      zo ja, dan niet nodig
    2   0S   6 Z K 0
    3   2A   0       A
    4   6A  25 Z E 1               LVC:= 0
    5   6T   0 Z F 0 0     =)      FRL met 2A 'LVC' A
    6   2S  25 Z K 0
    7   2A   0       A
    8   6T   0 Z F 0 0     =)      FRL met 4A 17 X1
    9   2S  26 Z K 0
   10   2A   0       A
   11   6T   0 Z F 0 0     =)      FRL met 4A 18 X1
   12   2T   9   X 0   E   =>      klaar
        DC D0
```

```
RLA      reservation local or value arrays                       KN0

aanroep                          6T 0 K N0 2  =)  RLV



          DA   0 K N 0       DI
    =)  0   2A   4 Z E 2   Z            VLAM = 0? dwz., geen arrays?
        1 Y 2T  10   X 0   E   ->       zo ja, dan klaar
        2   5A   4 Z E 2                VLAM:= 0
        3   2B  25 Z E 0                TLSC
        4   2S  161       A
        5   0LS 32767 X 0 B Z           TLI[TLSC - 1] = blokbeginmarker?
        6 Y 2S  32766 X 0 B             pak in TLI
        7 N 2S  32765 X 0 B             gedumpte NLSC
        8   0S  31 Z E 0                NLIB
        9   6S   7 Z E 2                RLAA:= NLIB + NLSC-van-blokbegin
       10   2B  30 Z E 0                NLSC
       11   0B  31 Z E 0                NLIB
       12   2T   3 K N 1 A     =>
5KN1 => 13   6S   8 Z E 0   P           CYCLUS OPBOUW STOFU VOOR VALUE ARRAYS
       14 N 0P   1   SS    E            als d26 = 0 of d26 = 1 maar d25 = 1
       15 Y 2T  31 K N 0 A     ->       dan geen value array
       16   6B   8 Z E 2                dump NLSC in RLAB
       17   0P   6   SS    P            d19 = 0? dwz., real?
       18   6T   0 Z R 0 0     =)       AVR
       19 Y 2A  92        A             zo ja, dan OPC van RVA
       20 N 2A  93        A             zo neen, dan OPC van IVA
       21   2S   0        A
       22   6T   0 Z F 0 0     =)       FRL met RVA of IVA
       23   2B   8 Z E 2                pak de in RLAB gedumpte NLSC
       24   2S  32767 X 0 B             pak ID uit NLI
       25   3LS 32767    A              schrap adresdeel
       26   3LS  3 Z K 0                d16 = 0 maken: non-formeel
       27   0S  23 Z E 1                PNLV als adres toevoegen
       28   6S  32767 X 0 B             ID klaar, naar NLI ermee
       29   2A   0   X 8 A              8 * 32
       30   4A  23 Z E 1                ter ophoging van PNLV: hoogstens 5
  15 -> 31   2S  32766 X 0 B            INW uit NLI                indices
          DC D0
```

KN1

```
            DA   0 K N 1       DI
         0   2LS  7        A Z          enkelwoordsnaam?
         1 Y 1B   2        A            NLSC passend aflagen
         2 N 1B   3        A
12KN0->  3 U 1B   7 Z E 2   Z           = RLAA?
         4 N 2S  32767 X 0 B            anders ID van volgende naam
         5 N 2T  13 K N 0 A     ->      op value array onderzoeken
         6   2B  30 Z E 0               NLSC
         7   0B  31 Z E 0               NLIB
  28 ->  8 U 1B   7 Z E 2   Z           = RLAA?
         9 Y 2T  29 K N 1 A     ->      zo ja, dan definitief klaar
        10   2S  32767 X 0 B P          RESERVERING LOCALE OF VALUE ARRAYS
        11 Y 2T  24 K N 1 A             als geen array dan skippen
        12 U 2LS 13 Z K 0   Z           d25 = 0? dwz., value array?
        13 N 3LS 13 Z K 0               anders d25
        14   3LS 28 Z K 0               maar in ieder geval d26
        15   6S  32767 X 0 B            schrappen uit ID in NLI
        16   6S   8 Z E 0               zet ID
        17   6B   8 Z E 2               dump NLSC in RLAB
        18   6T   0 Z R 0 0     =)      AVR
        19 Y 2A  95        A            als value dan OPC van VAP
        20 N 2A  94        A            als locaal dan OPC van LAP
        21   2S   0        A
        22   6T   0 Z F 0 0     =)      FRL met VAP of LAP
        23   2B   8 Z E 2               pak de in RLAB gedumpte NLSC
  11 -> 24   2S  32766 X 0 B            INW uit NLI
        25   2LS  7        A Z          enkelwoordsnaam?
        26 Y 1B   2        A            NLSC passend aflagen
        27 N 1B   3        A
        28   2T   8 K N 1 A     =>      ga volgende naam onderzoeken
   9 => 29   2A  29 Z E 0   Z           NFLA = 0?
        30 N 2B  22 Z E 0               zo neen, dan ID gaan
        31 N 0B  31 Z E 0               herstellen
             DC D0
```

KN2

```
    DA   0 K N 2      DI
0 N 2A   0   X 0 B              op grond van NLI[NID]
1 N 6A   8 Z E 0
2   2T  10   X 0   E   =>       klaar
    DC D0
```

```
DDEL  begin                                                                 LZ0


        DA   0 L Z 0        DI
=>  0   2B  25 Z E 0                 TLSC
    1   2S  32767 X 0 B              TLI[TLSC - 1]
    2   0LS 161      A Z             = blokbeginmarker?
    3 N 6T   0 K N 0 2      =)       zo neen, dan RLA
    4   2T  14 F E 0 A      =>       OH:= 0; FTD; terug naar basiscyclus
        DC D0
```

```
     SPS      start prescan                                        LE0


          DA   0 L E 0       DI
=>=>  0   2A    0      A
      1   6A  14 Z E 2            zet indicatie RNS maagdelijk
      2   6A  15 Z E 2            zet indicatie voor voorbereiding RNS
      3   2A  12       A
      4   6A  26   X 0            klasse 6 in neutrale toestand
      5   2T   6 L E 0   P   =>   zet vergunningen
      6   0X   7 L E 0            klasse 6 in LV, X1 doof
5 =>  7   6T   0 F T 0 2     =)   RND voor eerste begin
      8   2T   0 H K 0 A     =>   door naar PSP
          DC D0
```

```
TFO      test first occurrence                                  LFO

aanroep                      6T 0 L F0 0  =)  TFO



        DA   0 L F 0       DI
=)  0   2B  22 Z E 0             NID
    1   0B  31 Z E 0             NLIB
    2   2A   0   X 0 B           ID uit NLI
    3 U 2LA  0 Z K 0   Z         d15 = 0? dwz., eerder voorgekomen?
    4 Y 6A   8 Z E 0             zo ja, zet ID
    5 Y 2T   8   X 0   E   ->    en klaar
    6   4P         AS            VERDERE OPBOUW VOORLOPIGE ID:
    7   3LA  0 Z K 0             d15:= 0
    8   3LA 32767    A           clear adresgedeelte
    9   0A  13 Z K 0             voeg d25 toe: opc2 voor FLI-referentie
   10   0A  26 Z E 0             FLSC als adresgedeelte
   11   6A   0   X 0 B           vul ID in NLI in
   12   6A   8 Z E 0             zet ID
   13   2B  22 Z E 0             NID
   14 U 1B   9 Z E 2   P         > NLSC0? dwz., geen MCP?
   15   3B   1       A
   16   5B  26 Z E 0             FLSC:= FLSC + 1
   17 Y 2T   8   X 0   E   ->    klaar als geen MCP
   18   4P         SA            oude ID, = d18, d15, MCPnr
   19   0B  26 Z E 0             FLSC in kwestie
   20   0B  12 Z E 0             FLIB
   21   2T   0 F U 0 A     =>    door naar FFL
        DC D0
```

```
PST      procedure statement                                         LH0

aanroep                         6T 0 L H0 3  =)  PST



          DA    0 L H 0       DI
  =)  0   2A   18 Z E 0   Z           EFLA = 0?
      1 Y 6T    0 K N 0 2     =)      zo ja, dan RLA
      2   2B   22 Z E 0               NID
      3 U 1B   25 Z E 2   P           > NLSCop? dwz., geen standaardfunctie?
      4 N 2T    9 L H 0 A     ->      anders speciale behandeling
      5   2A   19 Z E 0   Z           FFLA = 0?
      6 Y 6T    0 L F 0 0     =)      zo ja, dan TF0
      7   6T    0 Z R 0 0     =)      BPR
      8   2T   11   X 0   E   =>      klaar
 4 => 9   0B   31 Z E 0               NLIB
     10   2S    0   X 0 B             pak ID uit NLI
     11   2LS 4095    A               isoleer 256 * OH + operatornummer
     12   6T    0 Z T 0 0     =)      FTL
     13   2A    9 Z E 0               DL
     14 U 1A   98       A Z           = (?
     15 Y 2A    1       A
     16 Y 6A   18 Z E 0               zo ja, dan EFLA:= 1
     17 Y 2T    4 E W 0 A     ->      en verder samen met DDEL (
     18   2T    8 Z S 0 A     =>      anders verder samen met POP
          DC D0
```

```
RFS      read FLEXOWRITER symbol                                   LK0

aanroep                          6T 0 L K0 14  =)  RFS



          DA   0 L K 0       DI
20-> =)  0   2Z   1   XP              heptade van band
         1   2A  18 Z E 2   Z         RFSB = 0?
         2 Y 2T  10 L K 0 A     ->    dan shift ongedefinieerd
         3   4P       SB
  14 ->  4   2S   0 L K 1 B P         pak tabel[heptade]; > +0?
         5 N 2T  22 L K 0 A           zo neen, dan uitzoeken
         6   0LA 124      A Z         RFSB = 124?, dwz., shift = uppercase?
         7 Y 3P   8   SS              zo ja, dan uitschuiven
         8 N 2LS 255      A           zo neen, dan isoleren
         9   2T  22   X 0   E   =>    klaar
   2 => 10   4P       SB
        11 U 0LS 62       A Z         heptade = 62? dwz., TAB?
        12 N 1S  16       A Z                   16?    SPATIE?
        13 N 1S  10       A Z                   26?      TWNR?
        14 Y 2T   4 L K 0 A     ->    zo ja, dan case-onafhankelijk
        15 U 0LS 96       A Z         heptade = 122? dwz., lower case?
        16 N 1S  98       A Z                   124?       upper case?
        17 N 0S  124      A Z                     0?            blank?
        18 Y 6B  18 Z E 2             zo ja, zet shift (ongedefinieerd)
        19 N 1S  127      A Z         heptade = 127? dwz., ERASE?
        20 Y 2T   0 L K 0 A     ->    zo ja, dan volgende heptade
        21   7Y  19   C 0             stop: shift niet gedefinieerd
   5 => 22   4P       SS    Z         symbool uit tabel = -0?
        23 Y 7Y  20   C 0             zo ja, dan stop: foute pariteit
        24 U 1LS  1       A Z         symbool uit tabel = -1?
        25 Y 7Y  21   C 0             zo ja, dan stop: ontoelaatbare ponsing
        26 U 1B  127      A Z         heptade = 127?
        27 N 6B  18 Z E 2             zo neen, dan zet shift
        28   2T   0 L K 0 A     =>    volgende heptade
          DC D0
```

LK1

```
      DA   0 L K 1      DN       SYMBOOL        TABELWAARDE
  0   -     2                    SHIFT ONGEDEFINIEERD
  1   + 19969                    OR  1            78     1
  2   + 16898                    *   2            66     2
  3   -     0                    foute pariteit
  4   + 18436                    =   4            72     4
  5   -     0                    foute pariteit
  6   -     0                    foute pariteit
  7   + 25863                    ]   7           101     7
  8   + 25096                    (   8            98     8
  9   -     0                    foute pariteit
 10   -     0                    foute pariteit
 11   -     1                    STOPCODE
 12   -     0                    foute pariteit
 13   -     1                    ontoelaatbare ponsing
 14   + 41635                    |   _           162   163
 15   -     0                    foute pariteit
 16   + 31611                    SPATIE          123   123
 17   -     0                    foute pariteit
 18   -     0                    foute pariteit
 19   + 17155                    /   3            67     3
 20   -     0                    foute pariteit
 21   + 23301                    ;   5            91     5
 22   + 25606                    [   6           100     6
 23   -     0                    foute pariteit
 24   -     0                    foute pariteit
 25   + 25353                    )   9            99     9
 26   + 30583                    TWNR            119   119
 27   -     0                    foute pariteit
 28   -     1                    ontoelaatbare ponsing
 29   -     0                    foute pariteit
 30   -     0                    foute pariteit
 31   -     1                    ontoelaatbare ponsing
      DC D0
```

	DA 0 L K 2	DN	SYMBOOL	TABELWAARDE	
0	+ 19712		and 0	77	0
1	- 0		foute pariteit		
2	- 0		foute pariteit		
3	+ 14365		T t	56	29
4	- 0		foute pariteit		
5	+ 14879		V v	58	31
6	+ 15136		W w	59	32
7	- 0		foute pariteit		
8	- 0		foute pariteit		
9	+ 15907		Z z	62	35
10	- 1		ontoelaatbare ponsing		
11	- 0		foute pariteit		
12	- 1		ontoelaatbare ponsing		
13	- 0		foute pariteit		
14	- 0		foute pariteit		
15	- 1		ontoelaatbare ponsing		
16	- 0		foute pariteit		
17	+ 17994		> <	70	74
18	+ 14108		S s	55	28
19	- 0		foute pariteit		
20	+ 14622		U u	57	30
21	- 0		foute pariteit		
22	- 0		foute pariteit		
23	+ 15393		X x	60	33
24	+ 15650		Y y	61	34
25	- 0		foute pariteit		
26	- 0		foute pariteit		
27	+ 30809		’ ten	120	89
28	- 0		foute pariteit		
29	- 1		ontoelaatbare ponsing		
30	+ 30326		TAB	118	118
31	- 0		foute pariteit		
	DC D0				

LK3

```
     DA   0 L K 3       DN        SYMBOOL          TABELWAARDE
 0   + 19521                      not -              76    65
 1   -     0                      foute pariteit
 2   -     0                      foute pariteit
 3   + 12309                      L   l              48    21
 4   -     0                      foute pariteit
 5   + 12823                      N   n              50    23
 6   + 13080                      O   o              51    24
 7   -     0                      foute pariteit
 8   -     0                      foute pariteit
 9   + 13851                      R   r              54    27
10   -     1                      ontoelaatbare ponsing
11   -     0                      foute pariteit
12   -     1                      ontoelaatbare ponsing
13   -     0                      foute pariteit
14   -     0                      foute pariteit
15   -     1                      ontoelaatbare ponsing
16   -     0                      foute pariteit
17   + 11795                      J   j              46    19
18   + 12052                      K   k              47    20
19   -     0                      foute pariteit
20   + 12566                      M   m              49    22
21   -     0                      foute pariteit
22   -     0                      foute pariteit
23   + 13337                      P   p              52    25
24   + 13594                      Q   q              53    26
25   -     0                      foute pariteit
26   -     0                      foute pariteit
27   + 31319                      ?   ,             122    87
28   -     0                      foute pariteit
29   -     1                      ontoelaatbare ponsing
30   -     1                      ontoelaatbare ponsing
31   -     0                      foute pariteit
     DC D0
```

	DA	0 L K 4	DN	SYMBOOL	TABELWAARDE	
0	-	0		foute pariteit		
1	+	9482		A a	37	10
2	+	9739		B b	38	11
3	-	0		foute pariteit		
4	+	10253		D d	40	13
5	-	0		foute pariteit		
6	-	0		foute pariteit		
7	+	11024		G g	43	16
8	+	11281		H h	44	17
9	-	0		foute pariteit		
10	-	0		foute pariteit		
11	+	31832		: .	124	88
12	-	0		foute pariteit		
13	-	1		ontoelaatbare ponsing		
14	-	1		ontoelaatbare ponsing		
15	-	0		foute pariteit		
16	+	31040		" +	121	64
17	-	0		foute pariteit		
18	-	0		foute pariteit		
19	+	9996		C c	39	12
20	-	0		foute pariteit		
21	+	10510		E e	41	14
22	+	10767		F f	42	15
23	-	0		foute pariteit		
24	-	0		foute pariteit		
25	+	11538		I i	45	18
26	-	2		LOWER CASE		
27	-	0		foute pariteit		
28	-	2		UPPER CASE		
29	-	0		foute pariteit		
30	-	0		foute pariteit		
31	-	2		ERASE		
	DC D0					

```
   BSM     bit string maker                                                LL0

   aanroepen                         6T  0 L L0 0  =)  BSM

                                     6T  7 L L1 0  =) voorbereiding BSM


         DA   0 L L 0       DI
=)  0    2A   0       A
    1    1P  10   SA                 bits naar A; S:= aantal aangeboden bits
    2    2B   0 L T 0                aantal bits in voorraad - 27
    3    6S   0 L T 0
    4    2S   0       A              schuif
    5    0P  10   AA                 aangeboden bits
    6    0P  27   SA  B              in goede positie
    7    0LA  1 L T 0                SA:= nieuwe voorraad bits
    8    4B   0 L T 0   P            nieuw aantal bits in voorraad > 27?
    9 N 6A    1 L T 0                zo neen, dan berg voorraad
   10 N 2T    8   X 0   E   ->       en klaar
   11    6S   1 L T 0                nieuwe voorraad:= kop van SA
   12    2S  27       A
   13    5S   0 L T 0                aantal bits:= aantal bits - 27
   14    2B  19 Z E 1                vulplaats
   15    6A   0   X 0 B              magazijn[vulplaats]:= staart van SA
   16    0B   1       A
   17    6B  19 Z E 1                vulplaats:= vulplaats + 1
   18    1B  20 Z E 1   Z            vulplaats = ledigplaats?
   19 N 2T    8   X 0   E   ->       zo neen, dan klaar
   20    2B  30 Z E 0                NLSC      OPSCHUIVEN VAN ALGOL-TEKST,
   21    0B  31 Z E 0                NLIB                  FLI, KLI EN NLI
   22    5P       BA
   23    1A   8       A
   24 U 0A   21 Z E 0   P            PLIB > NLIB + NLSC + 8?
   25 N 7Y   25   C 0                anders stop: programma te lang
   26    0A   8       A
   27    0A  20 Z E 1                ledigplaats
   28    7A   0   X 0                aantal:= NLIB + NLSC - ledigplaats
   29    2A   8       A
   30    4A  20 Z E 1                ledigplaats:= ledigplaats + 8
   31    4A  12 Z E 0                FLIB:= FLIB + 8
         DC D0
```

LL1

```
          DA   0 L L 1       DI
      0   4A  28 Z E 0               KLIB:= KLIB + 8
      1   4A  31 Z E 0               NLIB:= NLIB + 8
6 ->  2   1B   1       A             opschuifcyclus
      3   2A   0   X 0 B             8 plaatsen
      4   6A   8   X 0 B             omhoog
      5   4T   2 L L 1 0 P   ->
      6   2T   8   X 0   E   =>      klaar
7 =)  7   2S  27       A             VOORBEREIDING
      8   7S   0 L T 0               geen bits in voorraad
      9   2S   0       A
     10   6S   1 L T 0               clear voorraadwoord
     11   2S  18 Z E 1
     12   6S  19 Z E 1               vulplaats:= BIM
     13   2T   8   X 0   E   =>      klaar
          DC D0
```

```
CWD      code words                                                  LR0



     DA   0 L R 0       DN        OPCnr
 0   + 10624                       8
 1   +  6160                       9
 2   + 10625                      10
 3   + 10626                      11
 4   + 10627                      12
 5   +  7208                      13
 6   +  6161                      14
 7   + 10628                      15
 8   +  5124                      16
 9   +  7209                      17
10   +  6162                      18
11   +  7210                      19
12   +  7211                      20
13   + 10629                      21
14   + 10630                      22
15   + 10631                      23
16   + 10632                      24
17   + 10633                      25
18   + 10634                      26
19   + 10635                      27
20   + 10636                      28
21   + 10637                      29
22   +  6163                      30
23   +  7212                      31
24   + 10638                      32
25   +  4096                      33
26   +  4097                      34
27   +  7213                      35
28   + 10639                      36
29   + 10640                      37
30   + 10641                      38
31   +  7214                      39
     DC D0
```

```
     DA   0 L R 1       DN       OPCnr
 0   + 10642                     40
 1   + 10643                     41
 2   + 10644                     42
 3   + 10645                     43
 4   + 10646                     44
 5   + 10647                     45
 6   + 10648                     46
 7   + 10649                     47
 8   + 10650                     48
 9   + 10651                     49
10   + 10652                     50
11   + 10653                     51
12   + 10654                     52
13   + 10655                     53
14   + 10656                     54
15   + 10657                     55
16   +  5125                     56
17   + 10658                     57
18   +  5126                     58
19   + 10659                     59
20   + 10660                     60
21   +  7215                     61
     DC D0
```

```
ADC      address coder                                             LS0

aanroep                          6T 0 L S0 0  =)  ADC



     DA   0 L S 0      DI
=) 0   2A   8   X 0
   1   6A   5 L T 0             transporteer link
   2   6S   6 L T 0             red te coderen adres
   3   2LS 31       A           isoleer d4 - d0
   4 U 2LS 28       A Z         <= 3?
   5 N 0LS 6176     A           zo neen, bouw zelf het codewoord op
   6 Y 4P       SB              zo ja, dan
   7 Y 2S  25 L S 0 B           codewoord uit tabel halen
   8   6T   0 L L 0 0    =)     BSM met codewoord voor d4 - d0
   9   2S   6 L T 0             pak te coderen adres weer
  10   2LS 992      A           isoleer d9 - d5
  11   1P   5   SS              schuif uit
  12 U 1S   5       A P         > 5?
  13 Y 0LS 6176     A           dan zelf het codewoord opbouwen
  14 N 4P       SB              en anders
  15 N 2S  29 L S 0 B           codewoord uit tabel halen
  16   6T   0 L L 0 0    =)     BSM met codewoord voor d9 - d5
  17   2S   6 L T 0             pak te coderen adres weer
  18   2LS 31744    A Z         isoleer d14 - d10; = 0?
  19 N 1P  10   SS              zo neen, dan uitschuiven
  20 N 0LS 6176     A           en zelf het codewoord opbouwen
  21 Y 2S  1024     A           zo ja, dan codewoord pakken
  22   6T   0 L L 0 0    =)     BSM met codewoord voor d14 - d10
  23   2S   6 L T 0             herstel S
  24   2T   5 L T 0   E  =>     klaar
  25 DN +6176                   CODEWOORDEN     d4 - d0 = 0
  26   + 2048                                             1
  27   + 3074                                             2
  28   + 3075                                             3
  29   + 2048                                   d9 - d5 = 0
  30   + 4100                                             1
  31   + 4101                                             2
     DC D0
```

```
    DA   0 L S 1       DN
0   + 6179                      CODEWOORDEN    d9 - d5 = 3
1   + 4102                                               4
2   + 4103                                               5
    DC D0
```

```
      PLP      program loading program                                   RZ0

            DA    0 R Z 0       DI
9KW3 ->  0   2S    3 R E 0
         1   6S    1   X 0              telling:= RLSCE
         2   6T   19 R W 1 0     =)     voorbereiding RBS1
   4 ->  3   6T    5   D 1 0     =)     TPA?
         4 Y 1T    2       A     ->     zo ja, dan wacht
         5   6T    6 R L 0 3     =)     LIL voor RLI
         6   1S   89 S F 0   Z          laatste opdracht = OPC 96?
         7 N 7Y    5   C 3              anders stop: bitstroom ontspoord
         8   2S   11 R E 0              RLIB
         9   6T    0 R T 0 1     =)     TYP met RLIB
        10   2S   128      A
        11   6S    0   X 0              telling:= lengte MLI
  18 -> 12   2B    4 R K 0              CYCLUS MAAK COPIE VAN MLI
        13   0B    0   X 0
        14   2S   32767 X 0 B           MLI[i]
        15   2B    5 R K 0
        16   0B    0   X 0
        17   6S   32767 X 0 B           CRF[i]:= MLI[i]
        18   4T   12 R Z 0 0 P   ->
        19   2B    4 R K 0              MLIB
        20   6B    2 R E 0              geef opc3 een nieuwe betekenis
        21   2S    5 R E 0              FLIB
25      22   6S    6 R E 0              FLSC:= 0: vernietig FLI
1RZ1 -> 23   6T   29 R W 1 1     =)     voorbereiding RBS2
        24   6T    0 R U 0 4     =)     MCPL
  L4 => 25   2T   23 R Z 0 A     =>     volgende MCP lezen
  L4 => 26   2T    2 R Z 1 A     =>     doe test op einde
  L4 => 27   2B   13 R E 0              CYCLUS SKIP MCP: ledigplaats
  31 -> 28   1B    1       A
        29   2S    0   X 0 B            woord uit MCP-magazijn
        30   1S    4 R H 0   Z          een slotcombinatie?
        31 N 2T   28 R Z 0 A     ->     zo neen, dan verder skippen
             DC D0
```

```
                DA   0 R Z 1        DI
            0   6B  13 R E 0                  nieuwe ledigplaats
            1   2T  23 R Z 0 A      =>        volgende MCP lezen
26RZ0=>     2   2S  11 R Z 1 A
            3   2B  27   D16                  beginadres paragraaftabel
            4   6S   0   X 0 B                herdefinieer autostart 0
   6 ->     5   6T   5   D 1 0      =)        TPA?
            6 Y 1T   2       A      ->        zo ja, wacht dan
  14 ->     7   2S  14 R E 0   Z              aantal MCP's = 0?
            8 Y 2T  20 R Z 1 A      ->        dan is het objectprogramma klaar
            9   7Y   6   C 3                  anders stop: MCP-band inleggen
13         10   0Y  126  XS                   X1 doof
19->=>>    11   6T   5 R W 2 1      =)        voorbereiding RBS3
           12   6T   0 R U 0 4      =)        MCPL
  L4 =>    13   2T  11 R Z 1 A      =>        volgende MCP lezen
  L4 =>    14   2T   7 R Z 1 A      =>        doe test op einde
16,18=>    15   2Z   1   XP    Z              CYCLUS SKIP MCP: heptade = 0?
  l4       16 N 1T   2       A      ->        anders nog niet in blank
           17   2Z   1   XP    Z              heptade van band = 0?
           18 N 1T   4       A      ->        anders nog niet in blank
           19   2T  11 R Z 1 A      =>        volgende MCP lezen
   8 =>    20   2S  11 R E 0                  RLIB
           21   2B  27   D16                  beginadres paragraaftabel
           22   6S   0   X 0 B                herdefinieer autostart 0
           23   0Y   0   XS                   X1 horend
           24   2S  15   X 1                  MCPE
           25   6T   2 R T 0 1      =)        TYP met MCPE
           26   2S  15   X 1                  MCPE
           27   1S   3 R K 0                  GVC0
           28   6S   0   X 0                  telling:= lengte te clearen traject
           29   2S  15   X 1                  MCPE
           30   0S   0 R K 0                  clearopdracht opbouwen
           31   6S  24   X 1                  en wegschrijven
                DC D0
```

RZ2

```
             DA    0 R Z 2       DI
         0   3S    0       A
         1   3B    1       A
         2   2T   25   X 1 A     =>       naar cyclus clear werkruimte
24X2 =>  3   2A    5 R K 0                DIRECTIEF DW
         4   6A   28   X 0                transportadres:= CRFB
         5   2T   30   D 8 A     =>       verder alsof directief DB gelezen
             DC D0
```

```
RBW      read binary word                                       RF0

aanroep                         6T 0 R F0 2  =)  RBW



          DA    0 R F 0        DI
   =)  0   2A    0 S E 0   P            begint de codering met een bit = 0?
       1 Y 2T    4 R F 1 A      ->      dan een opdracht van het OPC-type
       2   2B    1       A              MASKERTYPE
       3   6T    0 R W 0 0      =)      RBS
       4   6T    0 R N 0 1      =)      ML voor functiegedeelte
       5   6S   10 R E 0                red dit
       6   6T    0 R Y 0 0              ADD voor adresgedeelte
       7   2S   10 R E 0                functiegedeelte
       8   2A    0       A              scheiden van
       9   0P   15   SA                 opc-nr
      10   0S    5 S E 0                voeg adresgedeelte toe
      11 U 2LA   1       A Z            opc = 0 of 2?
      12 Y 2T   21 R F 0 A      ->
      13 U 2LA   2       A Z            opc = 1?
      14 Y 0S    1 R E 0                dan RLIB bijtellen
      15 Y 2T   10   X 0   E    ->      en klaar
      16   2B    2 R E 0                bij opc = 3: pak KLIB of MLIB
      17 U 1B    4 R K 0   Z            is het MLIB? dwz., X3X bij MCP?
      18 N 0S    2 R E 0                zo neen, dan X3X in RLI, dus
      19 N 2T   10   X 0   E    ->      KLIB bijtellen en klaar
      20   2T   27 R F 0 A      =>      ga juiste adres uit MLI halen
12 => 21 U 2LA   2       A Z            opc = 0?
      22 Y 2T   10   X 0   E    ->      dan klaar
      23 U 2LS   2 R F 1   Z            bij opc = 2: als d17 <> 0
      24 N 3LS   2 R F 1                           dan d17:= 0,
      25 Y 0LS   3 R F 1                           anders d19:= 1
      26   2B    5 R E 0                FLIB voor X2X
20 -> 27   4P        SA
      28   2LA 32767     A              isoleer adresgedeelte
      29   6A   10 R E 0
      30   0B   10 R E 0
      31   3LS 32767     A              isoleer functiegedeelte
           DC D0
```

RF1

```
          DA   0 R F 1      DI
       0  0S   0   X 0 B          voeg FLI[adres] of MLI[adres] toe
       1  2T  10   X 0   E  =>    en klaar
       2  0A   0   X 0   P  |     d17
       3  0A   0   X 0 A    |     d19
1RF0 => 4  0P   1   AA    P        OPC-TYPE       beginbits 00?
       5 N 2T  14 R F 1 A   ->    anders > 5
       6  0P   1   AA    P        beginbits 000?
       7 Y 2B   4        A        zo ja, dan 0000 of 0001 voor 0 of 1
       8 N 2B   5        A        zo neen, dan 00100 t/m 00111 voor 2 t/m 5
       9  6T   0 R W 0 0    =)    RBS
      10 N 1S   2        A
22,26-> 11  4P       SB            haal opdracht
      12  2S   0 S F 0 B          uit OPC-tabel
      13  2T  10   X 0   E  =>    en klaar
   5 => 14  0P   1   AA    P       beginbits 010?
      15 N 2T  23 R F 1 A   ->    anders > 17
      16  0P   1   AA    P        beginbits 0100?
      17 Y 2B   6        A        dan 010000 t/m 010011 voor 6 t/m 9
      18 N 2B   7        A        anders 0101000 t/m 0101111 voor 10 t/m 17
      19  6T   0 R W 0 0    =)    RBS
      20 Y 1S  10        A
      21 N 1S  30        A
      22  2T  11 R F 1 A    =>    haal opdracht uit OPC-tabel en klaar
  15 => 23  2B  10        A        bij beginbits 011
      24  6T   0 R W 0 0    =)    RBS 0110000000 t/m 0111111111
      25  1S 366        A                        voor 18 t/m 145
      26  2T  11 R F 1 A    =>    haal opdracht uit OPC-tabel en klaar
          DC D0
```

```
TBV      test bitvoorraad                                              RH0

aanroep                        6T 0 R H0 0  =)  TBV



        DA   0 R H 0       DI
=)  0   2S   0 S E 0                bits in voorraad
    1   1S   4 R H 0   Z            = slotcombinatie?
    2 N 7Y  11   C 3                anders stop: decodering ontspoord
    3   2T   8   X 0   E   =)       klaar
    4   7Z   0   X 0       |        11 11110 00000 00000 00000 00000
        DC D0
```

constanten deel 2 RK0

```
    DA   0 R K 0       DI
 0  6S   0   X 0 C              clearopdracht
    DC D0
```

De inhoud van 1 R K0 t/m 5 R K0 staat vermeld bij de specifieke
constanten.

```
     LIL      lijst-inlezer                                          RL0

     aanroep                         6T 6 R L0 3  =)  LIL



          DA   0 R L 0       DI
  6 =>  0   6T   0 R F 0 2     =)       RBW                 BERGCYCLUS
 25 ->  1   2B   1 R E 0                vorm het
        2   0B   1   X 0                transportadres
        3 U 1B   6 R E 0   P            > FLIB + FLSCE?
        4 N 2T   8 R L 0 A     ->       anders FLI naar beneden schuiven
        5   6S   0   X 0 B              berg
    =)  6   4T   0 R L 0 1 E   ->       naar bergcyclus of
        7   2T  11   X 0   E   =>       klaar
  4 =>  8   2A   5 R E 0                FLIB        OPSCHUIVEN VAN FLI
        9   1A  13 R E 0                ledigplaats van RBS
       10   1A   1       A P            FLIB > ledigplaats + 1?
       11 N 7Y   2   C 3                anders stop: programma te lang
       12   6A   0 R E 0                schuifafastand
       13   5A   5 R E 0                nieuwe FLIB
       14   5A   6 R E 0                nieuwe (FLIB + FLSCE)
       15   2A   6 R E 0
       16   1A   5 R E 0
       17   6A   0   X 0                telling:= FLSCE
       18   2B  13 R E 0                ledigplaats
 24 -> 19   0B   0 R E 0                schuifafstand       SCHUIFCYCLUS
       20   2A   1   X 0 B
       21   1B   0 R E 0                over schuifafstand
       22   6A   1   X 0 B              omlaag
       23   0B   1       A
       24   4T  19 R L 0 0 P   ->
       25   2T   1 R L 0 A     =>       terug naar bergcyclys
            DC D0
```

```
  LLN      lees lengte of nummer                                      RR0

  aanroep                        6T 0 R R0 2  =)  LLN



         DA   0 R R 0      DI
=)  0    2B  13       A
    1    6T   0 R W 0 0    =)       RBS
    2 U  1S  7679     A P           eindmarker?
    3    2T  10   X 0      =>       klaar
         DC D0
```

```
HSC      haal symbool van CRF                                         RS0

aanroep                      6T 0 R S0 0  =)  HSC



        DA   0 R S 0       DI
=)  0   2A   0 R E 0             2 * haaladres
    1   1P   1   AA    P         even?
    2   2LA 32767   A            entier(haaladres)
    3   4P       AB
    4   2S   0   X 0 B           CRF[entier(haaladres)]
    5 Y 3P  13   SS              zo nodig hiervan de kop nemen
    6   2LS 8191    A            isoleer symbool van 13 bits
    7   2A   1      A
    8   4A   0 R E 0             haaladres:= haaladres + 1/2
    9 U 1S  7680    A Z          symbool = eindmarker (111 10000 00000)?
   10   2T   8   X 0       =>    klaar
        DC D0
```

```
TYP      typ S 32-tallig                                       RT0

aanroepen                          6T  0 R T0 1  =)  TYP met initialisatie

                                   6T  2 R T0 1  =)  TYP


         DA   0 R T 0       DI
=)  0    2A  19      A               zet schrijfmachine
    1    6Y   2   XP                 in kleine-letterstand
=)  2    2A  11      A               geef een
    3    6Y   2   XP                 TWNR
    4    4P       SA                 A:= 32 * 32 * a + 32 * b + c
    5    3P  10   SS                 S:= a
    6    0X  2176    A               S:= 32 * 68 * a + A
    7    4P       SA                 A:= 32 * 100 * a + 32 * b + c
    8    3P   5   SS                 S:= 100 * a + b
    9    0X  68      A               S:= 68 * 100 * a + 68 * b + A
   10    6T   0   D22 0     =)       typ S (= 10000 * a + 110 * b + c)
   11 DT AG6 NL2 SL2 SL2 XN
   12 DI 2T   9   X 0   E   =>       klaar
      DC D0
```

```
       RBS     read bits into S                                       RW0

       aanroepen                       6T  0 R W0 0  =)  RBS

                                       6T 19 R W1 0  =)  voorbereiding RBS1
                                       6T 29 R W1 1  =)  voorbereiding RBS2
                                       6T  5 R W2 1  =)  voorbereiding RBS3


            DA   0 R W 0       DI
     =)  0   2S   0       A
         1   2A   0 S E 0                pak bitvoorraad
         2   0P   0   SA  B              schuif gevraagde bits naar S
         3   6A   0 S E 0                berg nieuwe voorraad
         4   4B   1 S E 0 P              nieuwe 'ruimte' > 6?
 3RW1 -> 5 N 2T   8   X 0   E   ->       klaar als nog geen nieuwe heptade nodig
         6   6S   4 S E 0                red S
         7   2S   1       A
         8   4S   2 S E 0   P            heptadentelling:= heptadentelling + 1
         9   2T  12 R E 0       =>       switch (naar 10RW0 of 4 RW1)
    9 => 10   2Y   1   XP                heptade van band
         11 N 2T  26 R W 0 A    ->       als geen pariteitsonderzoek nodig
         12   4S   1 S E 0               'ruimte':= 'ruimte' + 1
         13   2S   3 S E 0               test de
         14   3P   4   SS                pariteit van
         15   0LS  3 S E 0               de vorige vier
         16   2LS 15       A             heptaden
         17   4P       SB
         18   2S  13515    A             01101 00110 01011
         19   1P   0   SS  B P           is de pariteit even?
         20 Y 7Y   9   C 3               dan stop: foute pariteit
16RW2=) 21   2S   3       A
         22   7S   2 S E 0               heptadentelling:= -3
         23   6A   3 S E 0               pariteitswoord:= gelezen heptade
         24   3P   1   AA                verwijder pariteitsbit
         25   2T  29 R W 0 A    =>
   11 => 26   4P       AS
         27   0LS  3 S E 0               pariteitswoord:= logische som van
18RW1-> 28   6S   3 S E 0                   pariteitswoord en gelezen heptade
   25 -> 29   2B   1 S E 0               schuif nieuwe heptade in goede positie
         30   6Y  32767 X 0 B            (2P 1- AA B)
         31   4A   0 S E 0               en voeg hem aan de voorraad toe
            DC D0
```

RW1

```
                DA   0 R W 1       DI
           0    2S   4 S E 0                herstel S
           1    1B   7       A
  28 ->    2    6B   1 S E 0   P            'ruimte':= 'ruimte' - 7, 'ruimte' > 6?
           3    2T   5 R W 0 A     =>       klaar met aanvulling voorraad
9RW0 =>    4    2A   0       A
           5 N  2T  15 R W 1 A     ->       als magazijnwoord nog niet leeg
           6    4S   1 S E 0                'ruimte':= 'ruimte' + 1
           7    2S   3       A
           8    7S   2 S E 0                heptadentelling:= -3
           9    2B  13 R E 0                ledigplaats
          10    1B   1       A
          11    6B  13 R E 0                ledigplaats:= ledigplaats - 1
          12    2S   0   X 0 B              S:= magazijn[ledigplaats]
          13    2B   6       A              eerste maal slechts 6 bits
          14    2T  17 R W 1 A     =>
   5 =>   15    2S   3 S E 0                pak magazijnwoord
          16    2B   7       A
  14 ->   17    0P   0   SA  B              schuif 'nieuwe heptade' naar A
          18    2T  28 R W 0 A     =>
     =)   19    2S   1 L T 0                VOORBEREIDING 1: voor RLI
          20    3B   0 L T 0                bouw
          21    0P   0   SS  B              bitvoorraad op
  31 =)   22    6S   0 S E 0
          23    1B   6       A              vorm 'ruimte'
          24    2S   0       A
          25    7S   2 S E 0                heptadentelling:= 0
          26    2S   4 R W 1 A
          27    6S  12 R E 0                zet switch op 4RW1
          28    2T   2 R W 1 A     =>       ga bitvoorraad aanvullen
     =)   29    2S   0       A              VOORBEREIDING 2: voor MCP's uit magazijn
          30    2B  27       A              'ruimte':= 33
          31    6T  22 R W 1 0     =)       doe stuk van voorbereiding 1
                DC D0
```

```
          DA   0 R W 2       DI
 3 ->  0   2B   1       A            CYCLUS SKIP BITS = 0
       1   6T   0 R W 0 0     =)     RBS voor 1 bit
       2   4P       SS    Z          = 0?
       3 Y 2T   0 R W 2 A     ->     dan herhalen
       4   2T   9   X 0   E   =>     klaar met voorbereiding
   =)  5   2S  10 R W 0 A            VOORBEREIDING 3: voor MCP's van band
       6   6S  12 R E 0              zet switch op 10RW0
       7   2S   0       A
       8   6S   0 S E 0              bitvoorraad:= 0
       9   2S  22       A
      10   6S   1 S E 0              'ruimte':= 28
12 -> 11   2Y   1   XP    Z          heptade van band = 0?
      12 Y 1T   2       A     ->     dan blank skippen
      13   OLA 30       A Z          eerste heptade = 30?
      14 N 7Y  10   C 3              anders stop
      15   2Y   1   XP               heptade van band
      16   6T  21 R W 0 0     =)     lees nog 3 heptades
      17   2T   0 R W 2 A     =>     en skip bits = 0
           DC D0
```

```
   MCPL     MCP-lezer                                                  RU0

   aanroep                          6T 0 R U0 4  =)  MCPL



        DA   0 R U 0       DI
 =)  0  6T   0 R R 0 2     =)      LLN voor lengte MCP
     1 Y 2T  18 R U 0 A     ->      als eindmarker
     2  6S   1   X 0                telling:= lengte MCP
     3  6T   0 R R 0 2     =)      LLN voor nummer MCP
     4  4P       SB
     5  0B   5 R K 0                CRFB
     6  2S   0   X 0 B Z            CRF[nummer MCP] = 0?
     7 Y 2T  19 R U 0 A     ->      dan MCP skippen
     8  6S   1 R E 0                geef opc1 goede betekenis
     9  6B   8 R E 0                dump (CRFB + nummer MCP)
    10  6T   6 R L 0 3     =)      LIL voor MCP
    11  6T   0 R H 0 0     =)      TBV
    12  2A   1       A
    13  5A  14 R E 0                aantal MCP's:= aantal MCP's - 1
    14  2A   0       A
    15  2B   8 R E 0                gedumpte (CRFB + nummer MCP)
    16  6A   0   X 0 B              CRF[nummer MCP]:= 0
    17  2T  12   X 0   E   =>      klaar met lezen
1 => 18  2A   1       A Z
7 -> 19 Y 2A   2       A
    20  4A  12   X 0                hoog lambda4 met 1 of 2 op
    21  2T  12   X 0   E   =>      klaar
        DC D0
```

```
ADD      address decoder                                          RY0

aanroep                         6T 0 R Y0 0  =)  ADD



         DA   0 R Y 0       DI
    =)  0   2S   0       A
        1   2A   0 S E 0   P          begint codering met een bit = 0?
        2 Y 2B   1       A            zo ja, dan 0 voor pentade 0
        3 N 2B   6       A            anders 100001 t/m 111111 voor 1 t/m 31
        4   6B   5 S E 0              onthoud aantal 'verbruikte' bits
        5   0P   0   SA  B            schuif juiste aantal bits naar S
        6 N 2LS 31       A            zo nodig d5 schoonmaken
        7   4P       AA    P          begint codering met een bit = 0?
        8 N 2T  20 R Y 0 A     ->     anders 2-de adrespentade = 3 of > 5
        9   0P   1   AA    P          beginbits 00?
       10 Y 3B   2       A            zo ja, dan 00 voor pentade 0
       11 N 3B   4       A            anders 0100 t/m 0111 voor 1, 2, 4, 5
       12   0P   6   SS  B            schuif 1-ste pentade over totaal
       13   5P       BB               5 plaatsen op, en het juiste aantal
       14   6Z  31   X 1 B            (0P 1- SA B)          bits uit A erbij
       15 Y 2T  23 R Y 0 A     ->     klaar met 2-de pentade = 0
       16 U 2LS  2       A Z          codering 0100 0f 0101?
       17 Y 1S   3       A            zo ja, dan 1 of 2 ervan maken
       18 N 0LS  2       A            anders van 0110 of 0111 nu 4 of 5 maken
       19   2T  23 R Y 0 A     =>
  8 => 20   0P   1   AA               gooi beginbit 1 weg
       21   2B   6       A
       22   6Z  31   X 1 B            (0P 1- SA B)   schuif 2-de pentade in S
15,19-> 23  4B   5 S E 0              tel aantal 'verbruikte' bits
       24   4P       AA    P          begint codering met bit = 0?
       25 N 2T   2 R Y 1 A     ->     anders 3-de adrespentade = 0 of > 3
       26   0P   1   AA    P          beginbits 00?
       27 Y 3B   2       A            zo ja, dan 00 voor pentade 1
       28 N 3B   3       A            anders 010 of 011 voor pentade 2 of 3
       29   0P   6   SS  B
       30   5P       BB
       31   6Z  31   X 1 B            (0P 1- SA B)
         DC D0
```

RY1

```
             DA   0 R Y 1       DI
         0 Y 0LS  1       A              bij codering 00 nog 1 optellen
         1   2T   5 R Y 1 A     =>
25RY0=>  2   0P   1   AA                 gooi beginbit 1 weg
         3   2B   6       A
         4   6Z  31   X 1 B              (0P 1- SA B)    schuif 3-de pentade in S
   1 ->  5   0B   5 S E 0                aantal 'verbruikte' bits
         6   6S   5 S E 0                gelezen adres in 5 S E0 afleveren
         7   2T   0 R W 0 A     =>       door naar RBS om 'verbruikte bits' te
             DC D0                                                  verwijderen
```

```
     ML       masker-lezer                                              RN0

     aanroep                          6T 0 R N0 0  =)  ML



          DA   0 R N 0       DI
      =)  0    2A   0 S E 0   P            begint codering met een bit = 0?
          1 N 2T    5 R N 0 A     ->       anders masker-nummer > 1
          2    2B   2       A              00 of 01 voor nummer 0 of 1
          3    6T   0 R W 0 0     =)       RBS
          4    2T  18 R N 0 A     =>       pak opdracht
    1 =>  5    0P   1   AA    P            beginbits 10?
          6 Y 2T   15 R N 0 A     ->       zo ja, dan 100 of 101 voor nummer 2 of 3
          7    2B   6       A              anders 110000 t/m 111111 voor 4 t/m 19
          8    6T   0 R W 0 0     =)       RBS
          9    1S  63       A Z            nummer = 19?
         10 N 0S   19       A
         11 N 2T   18 R N 0 A     ->       anders klaar met nummer
         12    6T   0 R Y 0 0     =)       ADD
         13    2S   5 S E 0                bij nummer 19 functiedeel als adres gecodeerd
         14    2T   9   X 0   E   =>       klaar
    6 => 15    2B   3       A
         16    6T   0 R W 0 0     =)       RBS
         17    0LS  6       A              maak er 2 of 3 van
 4,11 -> 18    4P       SB
         19    2S   0 S Z 0 B              pak functiedeel (plus opc) uit tabel
         20    2T   9   X 0   E   =>       klaar
               DC D0
```

```
MT       masker-tabel                                                SZ0



     DA   0 S Z 0       DN        opc   functie
 0   +   656                       0     2S  0    A
 1   + 14480                       3     2B  0    A
 2   + 10880                       2     2T  0 X0
 3   +  2192                       0     2B  0    A
 4   +   144                       0     2A  0    A
 5   + 10368                       2     2B  0 X0
 6   +  6800                       1     2T  0    A
 7   +     0                       0     0A  0 X0
 8   + 12304                       3     0A  0    A
 9   + 10883                       2   N 2T  0 X0
10   +  6288                       1     2B  0    A
11   +  4128                       1     0A  0 X0 B
12   +  8832                       2     2S  0 X0
13   +   146                       0   Y 2A  0    A
14   +   256                       0     4A  0 X0
15   +   134                       0   Y 2A  0 X0   P
16   +   402                       0   Y 6A  0    A
17   +  4144                       1     0A  0 X0 C
18   +    16                       0     0A  0    A
     DC D0
```

```
        CC       clear cycle                                                          X1



              DA  24   X 1       DI
  25 -> 24   0A   0   X 0                clearopdracht (zie 31 RZ1)
2RZ2 -> 25   4T  24   X 1 0 E   ->
  27 -> 26   6T   5   D 1 0     =>       TPA?
        27 Y 1T   2       A     ->       zo ja, wacht dan
        28   7Y   7   C 3                stop: klaar met vertalen
             DC D0
```

```
                                                                    X2



             DA  24   X 2       DI
26D17=> 24   2T   3 R Z 2 A     =>       behandeling directief DW
             DC D0
```

SW0

specifieke waarden d.d. 26-03-1997

```
DA   6 Z E 1       DN
+ 6783                      PLIE        6-19-31
DA   8 Z E 1
+  800                      TLIB        0-25- 0
DA  18 Z E 1
+  930                      BIM         0-29- 2
DA   0 Z E 2
+ 6880                      BOB         6-23- 0
DA   9 Z E 2
+    2                      NLSC0       0- 4-11
DA  17 Z E 2
+ 6944                      PNLIB       6-25- 0
DA  25 Z E 2
+    0                      NLSCop      0- 1- 1
DC D0


DA   1 R K 0       DN
+  928                      MCPB        0-29- 0
+10165                      KLIE        9-29-21
+  138                      GVC0        0- 4-10
+  800                      MLIB        0-25- 0
+  623                      CRFB        0-19-15
DC D0


DA  23   X 1       DN
+12256                      OCB6       11-31- 0
DC D0
```

SW1

```
DA  15   X19       DN        CRF
+ 245760           DI          30    0
2LS   20 X 0       DN        7680   20
+  15872                        1 7680
+  98306           DI          12    2
2LS   63 X 0       DN        7680   63
+  32256                        3 7680
+ 122884                       15    4
+  32256                        3 7680
+ 819205           DI         100    5
2LS  134 X 0       DN        7680  134
+  49176           DI           6   24
2LS   21 X 0       DN        7680   21
+ 204288           DI          24 7680
2LS 7680 X 0                 7680 7680 (eindmarker CRF)
DC D0
```

```
DA  18 Z E 1       DN
+   930                     BIM         0-29- 2
DA   9 Z E 2       DN
+    48                     NLSC0
DA  25 Z E 2       DN
+    31                     NLSCop
DA  6944 X 0       DN
+  27598040               ] read
+    265358                 d18 + 12*256 + 40 + 102
-         6               ] print
+  61580507               ]
+    265359                 d18 + 12*256 + 40 + 103
-  53284863               ] TAB
+    265360                 d18 + 12*256 + 40 + 104
-  19668591               ] NLCR
+    265361                 d18 + 12*256 + 40 + 105
-         0               ] SPACE
-  46937177               ]
+    265363                 d18 + 12*256 + 40 + 107
+  53230304               ] stop
+    265364                 d18 + 12*256 + 40 + 108
+  59085824               ] abs
+    265349                 d18 + 12*256 + 57 +  76
+  48768224               ] sign
+    265350                 d18 + 12*256 + 57 +  77
+  61715680               ] sqrt
+    265351                 d18 + 12*256 + 57 +  78
+  48838656               ] sin
+    265352                 d18 + 12*256 + 57 +  79
+  59512832               ] cos
+    265353                 d18 + 12*256 + 57 +  80
+  48922624               ] ln
+    265355                 d18 + 12*256 + 57 +  82
+  53517312               ] exp
+    265356                 d18 + 12*256 + 57 +  83
-       289               ] entier
+  29964985               ]
+    265357                 d18 + 12*256 + 57 +  84
-  29561343               ] SUM
+    294912                 d18 + d15 +  0
```

```
-  14789691              ] PRINTTEXT
-  15115337              ]
+    294913                d18 + d15 +  1
-  27986615              ] EVEN
+    294914                d18 + d15 +  2
-       325              ] arctan
+  21928153              ]
+    294915                d18 + d15 +  3
-  15081135              ] FLOT
+    294917                d18 + d15 +  5
-  14787759              ] FIXT
+    294918                d18 + d15 +  6
-      3610              ] ABSFIXT
-  38441163              ]
+    294936                d18 + d15 + 24
DC D0


DS
```

Appendix A

Compiler and run–time stops

During compilation of an ALGOL 60 program on the X1 the following stops could occur. (In case of a stop the stop number could be retrieved from the 10 least significant bits of the instruction register 'OR', which could be made visible on the operators console in a line of 27 light bulbs.) This list is taken from a user manual dated August 1st, 1962.

0– 1 Not interpretable.

0– 2 There occur too complicated constructions in the Algol program.

0– 3 The exponent of a constant is too large in absolute value.

0– 4 As stop: 0– 3.

0– 5 The store capacity available is too small for the Algol program.

0– 6 As stop: 0– 5.

0– 7 An identifier that has not been declared occurs in the Algol text.

0– 8 An unknown symbol occurs in the Algol text.

0– 9 End of PRESCAN.

0–10 As stop: 0– 1.

0–11 The symbol '|' is followed in the Algol text by a not permitted symbol.

0–12 A letter combination in the Algol text is underlined only in part.

0–13 A strange letter combination is underlined in the Algol text.

0–14 One of the symbols: ’ (accent), ” (apostrophe) or ? (question mark) occurs in the Algol text.

0–15 As stop 0– 5.

0–16 As stop 0– 5.

0–17 End of translation.

0–18 As stop 0– 5.

0–19 The shift on the paper tape is undefined after ‘tape feed’.

0–20 Parity error on the paper tape.

0–21 An unpermitted punching occurs on the paper tape.

During loading of the object tape the following stops could occur:

3– 1 The store capacity is too small for the program.

3– 2 As stop: 3– 1.

3– 3 Object program and machine do not fit.

3– 4 Stop after reading of FLI (the second part of the object tape).

3– 5 As stop: 3– 1.

3– 6 Stop after the reading of the cross–reference tape.

3– 7 Stop after reading of RLI (the first part of the object tape).

3– 8 Parity error in the tape.

3– 9 As stop: 3– 8.

3–10 As stop: 3– 8.

3–11 As stop: 3– 8.

3–12 As stop: 3– 8.

3–13 As stop: 3– 8.

During program execution the following stops could occur (taken from the user manual dated December 1st, 1962).

1– 1 An integer value exceeds the integer capacity.

1– 2 In the declaration of a dynamic array a lower bound is larger than the corresponding upper bound.

1– 3 On the input tape an unknown symbol is met by procedure 'read'.

1– 4 On the input tape a parity error is found by procedure 'read'.

1– 7 In function 'entier' an integer exceeds the integer capacity.

1– 8 At the operation '$\underline{:}$' (integer division) the two operands are not both of type integer.

1– 9 Program execution completed.

1–10 The actual parameter of 'XEEN' is not of integer type.

1–11 The actual parameter of procedure 'SPACE' is not of integer type.

1–12 A call of procedure 'stop' was executed.

1–17 On the input tape a shift definition is missing after 'tape feed'.

1–18 Procedure 'read' found the symbol 'STOP CODE' on the input tape (only allowed after the parts separator '?').

Appendix B

A sample ALGOL 60 program

The following ALGOL 60 program is taken from the PhD thesis of Zonneveld [11] and used for the measurements on the ALGOL 60 compiler for the X1 discussed in this report. It is printed in an non–original layout in order to improve readability and uses ‘ for $\vee$, ^ for $\wedge$, ~ for $\neg$, and % for $_{10}$.

Following the program text we give the output on the X1 console as produced during the compilation process.

```
_b_e_g_i_n  _c_o_m_m_e_n_t  JAZ164, R743, Outer Planets;

   _i_n_t_e_g_e_r  k,t; _r_e_a_l  a,k2,x;  _B_o_o_l_e_a_n  fi;
   _a_r_r_a_y  y,ya,z,za[1:15],m[0:5],e[1:60],d[1:33];

   _r_e_a_l  _p_r_o_c_e_d_u_r_e  f(k);  _i_n_t_e_g_e_r  k;
   _b_e_g_i_n  _i_n_t_e_g_e_r  i,j,i3,j3;  _r_e_a_l  p;
     _o_w_n  _r_e_a_l  _a_r_r_a_y  d[1:5,1:5],r[1:5];
     _i_f  k |= 1  _t_h_e_n  _g_o_t_o  A;
     _f_o_r  i:= 1  _s_t_e_p  1  _u_n_t_i_l  4  _d_o
     _b_e_g_i_n  i3:= 3*i;
       _f_o_r  j:= i+1  _s_t_e_p  1  _u_n_t_i_l  5  _d_o
       _b_e_g_i_n  j3:= 3*j;
         p:= (y[i3-2] - y[j3-2])|^2 + (y[i3-1] - y[j3-1])|^2 +
             (y[i3] - y[j3])|^2;
         d[i,j]:= d[j,i]:= 1/p/sqrt(p)
       _e_n_d
     _e_n_d ;
```

```
  _f_o_r  i:= 1  _s_t_e_p  1  _u_n_t_i_l  5  _d_o
  _b_e_g_i_n  i3:= 3*i; d[i,i]:= 0;
      p:= y[i3-2]|^2 + y[i3-1]|^2 + y[i3]|^2;
      r[i]:= 1/p/sqrt(p)
  _e_n_d ;
A: i:= (k - 1) _: 3 + 1;
   f:= k2 * (- m[0] * y[k] * r[i] +
   SUM(j,1,5,m[j]*((y[3*(j-i)+k]-y[k])*d[i,j]-y[3*(j-i)+k]*r[j])))
_e_n_d  f;

_p_r_o_c_e_d_u_r_e  RK3n(x,a,b,y,ya,z,za,fxyj,j,e,d,fi,n);
_v_a_l_u_e  b,fi,n;  _i_n_t_e_g_e_r  j,n;  _r_e_a_l  x,a,b,fxyj;
_B_o_o_l_e_a_n  fi;  _a_r_r_a_y  y,ya,z,za,e,d;
_b_e_g_i_n  _i_n_t_e_g_e_r  jj;
  _r_e_a_l  xl,h,hmin,int,hl,absh,fhm,discry,discrz,toly,tolz,mu,mu1,fhy,fhz;
  _B_o_o_l_e_a_n  last,first,reject;
  _a_r_r_a_y  yl,zl,k0,k1,k2,k3,k4,k5[1:n],ee[1:4*n];
  _i_f  fi
  _t_h_e_n  _b_e_g_i_n  d[3]:= a;
              _f_o_r  jj:= 1  _s_t_e_p  1  _u_n_t_i_l  n  _d_o
              _b_e_g_i_n  d[jj+3]:= ya[jj]; d[n+jj+3]:= za[jj]  _e_n_d
            _e_n_d ;
  d[1]:= 0; xl:= d[3];
  _f_o_r  jj:= 1  _s_t_e_p  1  _u_n_t_i_l  n  _d_o
  _b_e_g_i_n  yl[jj]:= d[jj+3]; zl[jj]:= d[n+jj+3]  _e_n_d ;
  _i_f  fi  _t_h_e_n  d[2]:= b - d[3];
  absh:= h:= abs(d[2]);
  _i_f  b - xl < 0  _t_h_e_n  h:= - h;
  int:= abs(b - xl); hmin:= int * e[1] + e[2];
  _f_o_r  jj:= 2  _s_t_e_p  1  _u_n_t_i_l  2*n  _d_o
  _b_e_g_i_n  hl:= int * e[2*jj-1] + e[2*jj];
    _i_f  hl < hmin  _t_h_e_n  hmin:= hl
  _e_n_d ;
  _f_o_r  jj:= 1  _s_t_e_p  1  _u_n_t_i_l  4*n  _d_o  ee[jj]:= e[jj]/int;
  first:= reject:=  _t_r_u_e ;
  _i_f  fi
  _t_h_e_n  _b_e_g_i_n  last:=  _t_r_u_e ;  _g_o_t_o  step  _e_n_d ;
test: absh:= abs(h);
  _i_f  absh < hmin
  _t_h_e_n  _b_e_g_i_n  h:=  _i_f  h > 0  _t_h_e_n  hmin  _e_l_s_e  - hmin;
```

```
              absh:= hmin
            _e_n_d ;
  _i_f  h _> b - xl _= h _> 0
  _t_h_e_n  _b_e_g_i_n  d[2]:= h; last:=  _t_r_u_e ;
              h:= b - xl; absh:= abs(h)
            _e_n_d
  _e_l_s_e  last:=  _f_a_l_s_e ;
step:  _i_f  reject
  _t_h_e_n  _b_e_g_i_n  x:= xl;
              _f_o_r  jj:= 1  _s_t_e_p  1  _u_n_t_i_l  n  _d_o
              y[jj]:= yl[jj];
              _f_o_r  j:= 1  _s_t_e_p  1  _u_n_t_i_l  n  _d_o
              k0[j]:= fxyj * h
            _e_n_d
  _e_l_s_e  _b_e_g_i_n  fhy:= h/hl;
              _f_o_r  jj:= 1  _s_t_e_p  1  _u_n_t_i_l  n  _d_o
              k0[jj]:= k5[jj] * fhy
            _e_n_d ;
  x:= xl + .27639 32022 50021 * h;
  _f_o_r  jj:= 1  _s_t_e_p  1  _u_n_t_i_l  n  _d_o
  y[jj]:= yl[jj] + (zl[jj] * .27639 32022 50021 +
                  k0[jj] * .03819 66011 25011) * h;
  _f_o_r  j:= 1  _s_t_e_p  1  _u_n_t_i_l  n  _d_o  k1[j]:= fxyj * h;
  x:= xl + .72360 67977 49979 * h;
  _f_o_r  jj:= 1  _s_t_e_p  1  _u_n_t_i_l  n  _d_o
  y[jj]:= yl[jj] + (zl[jj] * .72360 67977 49979 +
                  k1[jj] * .26180 33988 74989) * h;
  _f_o_r  j:= 1  _s_t_e_p  1  _u_n_t_i_l  n  _d_o  k2[j]:= fxyj * h;
  x:= xl + h * .5;
  _f_o_r  jj:= 1  _s_t_e_p  1  _u_n_t_i_l  n  _d_o
  y[jj]:= yl[jj] + (zl[jj] * .5 +
                  k0[jj] * .04687 5 +
                  k1[jj] * .07982 41558 39840 -
                  k2[jj] * .00169 91558 39840) * h;
  _f_o_r  j:= 1  _s_t_e_p  1  _u_n_t_i_l  n  _d_o  k4[j]:= fxyj * h;
  x:=  _i_f  last  _t_h_e_n  b  _e_l_s_e  xl + h;
  _f_o_r  jj:= 1  _s_t_e_p  1  _u_n_t_i_l  n  _d_o
  y[jj]:= yl[jj] + (zl[jj] +
                  k0[jj] * .30901 69943 74947 +
                  k2[jj] * .19098 30056 25053) * h;
```

```
  _f_o_r  j:= 1  _s_t_e_p  1  _u_n_t_i_l  n  _d_o  k3[j]:= fxyj * h;
  _f_o_r  jj:= 1  _s_t_e_p  1  _u_n_t_i_l  n  _d_o
  y[jj]:= yl[jj] + (zl[jj] +
                  k0[jj] * .08333 33333 33333 +
                  k1[jj] * .30150 28323 95825 +
                  k2[jj] * .11516 38342 70842) * h;
  _f_o_r  j:= 1  _s_t_e_p  1  _u_n_t_i_l  n  _d_o  k5[j]:= fxyj * h;
  reject:=  _f_a_l_s_e ; fhm:= 0;
  _f_o_r  jj:= 1  _s_t_e_p  1  _u_n_t_i_l  n  _d_o
  _b_e_g_i_n
    discry:= abs((- k0[jj] * .5 + k1[jj] * 1.80901 69943 74947 +
                  k2[jj] * .69098 30056 25053 - k4[jj] * 2) * h);
    discrz:= abs((k0[jj] - k3[jj]) * 2 - (k1[jj] + k2[jj]) * 10 +
                k4[jj] * 16 + k5[jj] * 4);
    toly:= absh * (abs(zl[jj]) * ee[2*jj-1] + ee[2*jj]);
    tolz:= abs(k0[jj]) * ee[2*(jj+n)-1] + absh * ee[2*(jj+n)];
    reject:= discry > toly ‘ discrz > tolz ‘ reject;
    fhy:= discry/toly; fhz:= discrz/tolz;
    _i_f  fhz > fhy  _t_h_e_n  fhy:= fhz;
    _i_f  fhy > fhm  _t_h_e_n  fhm:= fhy
  _e_n_d ;
  mu:= 1/(1 + fhm) + .45;
  _i_f  reject
  _t_h_e_n  _b_e_g_i_n  _i_f  absh _< hmin
            _t_h_e_n  _b_e_g_i_n  d[1]:= d[1] + 1;
                      _f_o_r  jj:= 1  _s_t_e_p  1  _u_n_t_i_l  n  _d_o
                      _b_e_g_i_n  y[jj]:= yl[jj];
                        z[jj]:= zl[jj]
                      _e_n_d ;
                      first:= _t_r_u_e ;  _g_o_t_o  next
                    _e_n_d ;
            h:= mu * h;  _g_o_t_o  test
          _e_n_d  rej;
  _i_f  first
  _t_h_e_n  _b_e_g_i_n  first:= _f_a_l_s_e ;  hl:= h; h:= mu * h;
            _g_o_t_o  acc
          _e_n_d ;
  fhy:= mu * h/hl + mu - mu1; hl:= h; h:= fhy * h;
acc: mu1:= mu;
  _f_o_r  jj:= 1  _s_t_e_p  1  _u_n_t_i_l  n  _d_o
```

```
  z[jj]:= zl[jj] + (k0[jj] + k3[jj]) * .08333 33333 33333 +
                   (k1[jj] + k2[jj]) * .41666 66666 66667;
next:  _i_f b |= x
  _t_h_e_n  _b_e_g_i_n  xl:= x;
             _f_o_r  jj:= 1  _s_t_e_p  1  _u_n_t_i_l  n  _d_o
             _b_e_g_i_n  yl[jj]:= y[jj]; zl[jj]:= z[jj]  _e_n_d ;
             _g_o_t_o  test
           _e_n_d ;
  _i_f ~ last  _t_h_e_n  d[2]:= h;
  d[3]:= x;
  _f_o_r  jj:= 1  _s_t_e_p  1  _u_n_t_i_l  n  _d_o
  _b_e_g_i_n  d[jj+3]:= y[jj]; d[n+jj+3]:= z[jj]  _e_n_d
_e_n_d  RK3n;

_p_r_o_c_e_d_u_r_e  TYP(x);  _a_r_r_a_y  x;
_b_e_g_i_n  _i_n_t_e_g_e_r  k;
  NLCR; PRINTTEXT(|<T = |>); ABSFIXT(7,1,t+a); NLCR; NLCR;
  _f_o_r  k:= 1  _s_t_e_p  1  _u_n_t_i_l  5  _d_o
  _b_e_g_i_n  _i_f  k=1  _t_h_e_n  PRINTTEXT(|<J   |>)  _e_l_s_e
    _i_f  k=2  _t_h_e_n  PRINTTEXT(|<S   |>)  _e_l_s_e
    _i_f  k=3  _t_h_e_n  PRINTTEXT(|<U   |>)  _e_l_s_e
    _i_f  k=4  _t_h_e_n  PRINTTEXT(|<N   |>)  _e_l_s_e
                        PRINTTEXT(|<P   |>);
    FIXT(2,9,x[3*k-2]); FIXT(2,9,x[3*k-1]); FIXT(2,9,x[3*k]);
    NLCR
  _e_n_d
_e_n_d  TYP;

a:= read;
_f_o_r  k:= 1  _s_t_e_p  1  _u_n_t_i_l  15  _d_o
_b_e_g_i_n  ya[k]:= read; za[k]:= read  _e_n_d ;
_f_o_r  k:= 0  _s_t_e_p  1  _u_n_t_i_l  5  _d_o  m[k]:= read;
k2:= read; e[1]:= read;
_f_o_r  k:= 2  _s_t_e_p  1  _u_n_t_i_l  60  _d_o  e[k]:= e[1];
NLCR; PRINTTEXT(|<JAZ164, R743, Outer Planets|>); NLCR; NLCR;
_f_o_r  k:= 1  _s_t_e_p  1  _u_n_t_i_l  15  _d_o
_b_e_g_i_n  FLOT(12,ya[k]); FLOT(12,za[k]); NLCR  _e_n_d ;
_f_o_r  k:= 0  _s_t_e_p  1  _u_n_t_i_l  5  _d_o
_b_e_g_i_n  NLCR; FLOT(12,m[k])  _e_n_d ;
NLCR; NLCR; FLOT(12,k2);
```

```
   NLCR; NLCR; PRINTTEXT(|<eps = |>); FLOT(2,e[1]); NLCR;
   t:= 0; TYP(ya); fi:=  _t_r_u_e ;
   _f_o_r  t:= 500,1000  _d_o
   _b_e_g_i_n  RK3n(x,0,t,y,ya,z,za,f(k),k,e,d,fi,15);
     fi:=  _f_a_l_s_e ;  TYP(y)
   _e_n_d
_e_n_d
```

Here follows the output on the console typewriter during compilation and program loading:

```
f        00 00 02
A        00 07 25
RK3n     00 11 04
test     00 22 28
step     00 25 13
acc      01 25 21
next     01 27 28
TYP      02 00 02
  7 11  0

  7  1 15
```

Appendix C

The OPC table

Below follows a list of all OPCs as documented in [5]. OPC 81, originally in use for arctan, became obsolete after replacement of the complex routine for it by an MCP using another algorithm.

8	ETMR	EXTRANSMARK RESULT
9	ETMP	EXTRANSMARK PROCEDURE
10	FTMR	FORMTRANSMARK RESULT
11	FTMP	FORMTRANSMARK PROCEDURE
12	RET	RETURN
13	EIS	END OF IMPLICIT SUBROUTINE
14	TRAD	TAKE REAL ADDRESS DYNAMIC
15	TRAS	TAKE REAL ADDRESS STATIC
16	TIAD	TAKE INTEGER ADDRESS DYNAMIC
17	TIAS	TAKE INTEGER ADDRESS STATIC
18	TFA	TAKE FORMAL ADDRESS
19	FOR0	
20	FOR1	
21	FOR2	
22	FOR3	
23	FOR4	

24	FOR5	
25	FOR6	
26	FOR7	
27	FOR8	
28	GTA	GOTO ADJUSTMENT
29	SSI	STORE SWITCH INDEX
30	CAC	COPY BOOLEAN ACC. INTO CONDITION
31	TRRD	TAKE REAL RESULT DYNAMIC
32	TRRS	TAKE REAL RESULT STATIC
33	TIRD	TAKE INTEGER RESULT DYNAMIC
34	TIRS	TAKE INTEGER RESULT STATIC
35	TFR	TAKE FORMAL RESULT
36	ADRD	ADD REAL DYNAMIC
37	ADRS	ADD REAL STATIC
38	ADID	ADD INTEGER DYNAMIC
39	ADIS	ADD INTEGER STATIC
40	ADF	ADD FORMAL
41	SURD	SUBTRACT REAL DYNAMIC
42	SURS	SUBTRACT REAL STATIC
43	SUID	SUBTRACT INTEGER DYNAMIC
44	SUIS	SUBTRACT INTEGER STATIC
45	SUF	SUBTRACT FORMAL
46	MURD	MULTIPLY REAL DYNAMIC
47	MURS	MULTIPLY REAL STATIC
48	MUID	MULTIPLY INTEGER DYNAMIC
49	MUIS	MULTIPLY INTEGER STATIC
50	MUF	MULTIPLY FORMAL
51	DIRD	DIVIDE REAL DYNAMIC
52	DIRS	DIVIDE REAL STATIC
53	DIID	DIVIDE INTEGER DYNAMIC
54	DIIS	DIVIDE INTEGER STATIC
55	DIF	DIVIDE FORMAL

56	IND	INDEXER
57	NEG	INVERT SIGN ACCUMULATOR
58	TAR	TAKE RESULT
59	ADD	ADD
60	SUB	SUBTRACT
61	MUL	MULTIPLY
62	DIV	DIVIDE
63	IDI	INTEGER DIVISION
64	TTP	TO THE POWER
65	MOR	MORE $>$
66	LST	AT LEAST $\geq$
67	EQU	EQUAL $=$
68	MST	AT MOST $\leq$
69	LES	LESS $<$
70	UQU	UNEQUAL $\neq$
71	NON	NON $\neg$
72	AND	AND $\wedge$
73	OR	OR $\vee$
74	IMP	IMPLIES $\rightarrow$
75	QVL	EQUIVALENT $\equiv$
76	abs	
77	sign	
78	sqrt	
79	sin	
80	cos	
82	ln	
83	exp	
84	entier	
85	ST	STORE
86	STA	STORE ALSO
87	STP	STORE PROCEDURE VALUE
88	STAP	STORE ALSO PROCEDURE VALUE

89	SCC	SHORT CIRCUIT
90	RSF	REAL ARRAYS STORAGE FUNCTION FRAME
91	ISF	INTEGER ARRAYS STORAGE FUNCTION FRAME
92	RVA	REAL VALUE ARRAY STORAGE FUNCTION FRAME
93	IVA	INTEGER VALUE ARRAY STORAGE FUNCTION FRAME
94	LAP	LOCAL ARRAY POSITIONING
95	VAP	VALUE ARRAY POSITIONING
96	START	start of the object program
97	STOP	end of the object program
98	TFP	TAKE FORMAL PARAMETER
99	TAS	TYPE ALGOL SYMBOL
100	OBC6	OUTPUT BUFFER CLASS 6
101	FLOATER	
102	read	
103	print	
104	TAB	
105	NLCR	
106	XEEN	
107	SPACE	
108	stop	
109	P21	

Appendix D

The compact code

The compact code of the object program in the ALD7 and the load–and–go versions of the compiler (cf. Chapter 6) is given in two tables. The first one gives the encoding of OPCs with OPC number at least 8, the second one the encoding of 19 OPC–instruction combinations.

length	codebits	OPC–nr	acronym	full name
4	0000	33	TIRD	take integer result dynamic
4	0001	34	TIRS	take integer result static
5	00100	16	TIAD	take integer address dynamic
5	00101	56	IND	indexer
5	00110	58	TAR	take result
5	00111	85	ST	store
6	010000	9	ETMP	extransmark procedure
6	010001	14	TRAD	take real address dynamic
6	010010	18	TFA	take formal address
6	010011	30	CAC	copy boolean acc. into condition
7	0101000	13	EIS	end of implicit subroutine
7	0101001	17	TIAS	take integer address static
7	0101010	19	FOR0	for0
7	0101011	20	FOR1	for1
7	0101100	31	TRRD	take real result dynamic
7	0101101	35	TFR	take formal result
7	0101110	39	ADIS	add integer static
7	0101111	61	MUL	multiply
10	0110000000	8	ETMR	extransmark result
10	0110000001	10	FTMR	formtransmark result
10	0110000010	11	FTMP	formtransmark procedure
10	0110000011	12	RET	return
10	0110000100	15	TRAS	take real address static
10	0110000101	21	FOR2	for2
10	...	...		
10	0110001101	29	SSI	store switch index
10	0110001110	32	TRRS	take real result static
10	0110001111	36	ADRD	add real dynamic
10	0110010000	37	ADRS	add real static
10	0110010001	38	ADID	add integer dynamic
10	0110010010	40	ADF	add formal
10	...	...		
10	0110100001	55	DIF	divide formal
10	0110100010	57	NEG	invert sign accumulator
10	0110100011	59	ADD	add
10	0110100100	60	SUB	subtract
10	0110100101	62	DIV	divide
10	...	...		
10	0110111011	84	entier	entier
10	0110111101	86	STA	store also
10	...	...		
10	0111010100	109	P21	p21

length	codebits	OPC	X1–instruction
3	100	0	2S 0 A
3	101	3	2B 0 A
4	1100	2	2T 0
4	1101	0	2B 0 A
7	1110000	0	2A 0 A
7	1110001	2	2B 0
7	1110010	1	2T 0 A
7	1110011	0	0A 0
7	1110100	3	0A 0 A
7	1110101	2	N 2T 0
7	1110110	1	2B 0 A
7	1110111	1	0A 0 B
7	1111000	2	2S 0
7	1111001	0	Y 2A 0 A
7	1111010	0	4A 0
7	1111011	0	Y 2A 0 P
7	1111100	0	Y 6A 0 A
7	1111101	1	0A 0 C
7	1111110	0	0A 0 A
7	1111111	all other cases	

Bibliography

[1] E.W. Dijkstra. *Communication with an automatic computer.*
PhD Thesis University of Amsterdam, 1959.

[2] E.W. Dijkstra. *Cursus programmeren in ALGOL 60.*
Mathemathisch Centrum, 1960.

[3] E.W. Dijkstra. *A primer of ALGOL 60: report on the algorithmic language ALGOL 60.*
Academic Press, London 1962.

[4] E.W. Dijkstra. *Operating experience with ALGOL 60.*
The Computer Journal, **5** (1962), 125–127.

[5] E.W. Dijkstra. *Objectprogramma, gegenereerd door de M.C. vertaler.*
Mathematisch Centrum report MR 55, 1963.

[6] E.W. Dijkstra. *An ALGOL-60 translator for the X1.*
Annual Rev. in Autom. Progr., **3** (1963), 329–345.

[7] E.W. Dijkstra. *Making a translator for ALGOL-60.*
Annual Rev. in Autom. Progr., **3** (1963), 347–356.

[8] B.J. Loopstra. *The X–1 Computer.*
The Computer Journal, **2** (1958) 39–43.

[9] J.W.Backus et.al. *Report on the Algorithmic Language ALGOL60.*
Regnecentralen, Copenhagen, 1960

[10] P.Naur (ed.) *Revised Report on the Algorithmic Language ALGOL60.*
Regnecentralen, Copenhagen, 1962

[11] J.A. Zonneveld. *Automatic Numerical Integration.* PhD Thesis University of Amsterdam, 1964.

[12] J.A.Th.M. van Berckel and B.J. Mailloux. *Some ALGOL plotting procedures.* Mathematisch Centrum report MR 73, 1965.