



Centrum Wiskunde & Informatica

REPORTRAPPORT

SEN

Software Engineering



Software ENgineering

A comparison between the ALGOL 60 implementations
on the Electrologica X1 and the Electrologica X8

F.E.J. Kruseman Aretz

REPORT SEN-E0801 SEPTEMBER 2008

A comparison between the ALGOL 60 implementations on the Electrologica X1 and the Electrologica X8

ABSTRACT

We compare two ALGOL 60 implementations, both developed at the Mathematical Centre in Amsterdam, forerunner of the CWI. They were designed for Electrologica hardware, the EL X1 from 1958 and the EL X8 from 1965. The X1 did not support ALGOL 60 implementation at all. Its ALGOI 60 system was designed by Dijkstra and Zonneveld and completed in 1960. Although developed as an academic exercise it soon was heavily used and appreciated. The X8, successor of the X1 and (almost) upwards compatible to it, had a number of extensions chosen specifically to support ALGOL 60 implementation. The ALGOL 60 system, developed by Nederkoorn and Kruseman Aretz, was completed even before the first delivery of an X8. In this document we describe the two systems, demonstrating the progress in both hardware and software in a relatively short period.

2000 Mathematics Subject Classification: 01-08; 68-03; 68N20

1998 ACM Computing Classification System: K.2; D.3.4

Keywords and Phrases: History of Computing; ALGOL 60 compiler; Electrologica X1; Electrologica X8

Reprocenter Automatische Opdrachtbon

Voor het maken van een CWI-rapport

Afdeling: SEN 1

Contactpersoon: Paul.Klint@cwi.nl

Titel: A comparison between the ALGOL 60 implementations on the Electrologica X1 and the Electrologica X8

Aanmaakdatum bon: 23-09-2008

Oplage auteur: 15

In de database:

Rapportnummer: SEN-E0801

Titel: A comparison between the ALGOL 60 implementations on the Electrologica X1 and the Electrologica X8

Jaar: 2008

CR cat.: K.2; D.3.4

AM. cat: 01-08; 68-03; 68N20

ISSN: 1386-369X

Pagina's: 182

Size:

Department: SEN 1

Authors: F.E.J. Kruseman Aretz

Affiliaties: Professor Emeritus Technical University Eindhoven

Abstract: We compare two ALGOL 60 implementations, both developed at the Mathematical Centre in Amsterdam, forerunner of the CWI. They were designed for Electrologica hardware, the EL X1 from 1958 and the EL X8 from 1965. The X1 did not support ALGOL 60 implementation at all. Its ALGOL 60 system was designed by Dijkstra and Zonneveld and completed in 1960. Although developed as an academic exercise it soon was heavily used and appreciated. The X8, successor of the X1 and (almost) upwards compatible to it, had a number of extensions chosen specifically to support ALGOL 60 implementation. The ALGOL 60 system, developed by Nederkoorn and Kruseman Aretz, was completed even before the first delivery of an X8. In this document we describe the two systems, demonstrating the progress in both hardware and software in a relatively short period.

Keywords: History of Computing; ALGOL 60 compiler; Electrologica X1; Electrologica X8

Opmerkingen: This report is historical in nature. The author is professor emeritus at Technical University Eindhoven.

A comparison between
the ALGOL 60 implementations on
the Electrologica X1 and the Electrologica X8

F.E.J. Kruseman Aretz

September 16, 2008

ABSTRACT

We compare two ALGOL 60 implementations, both developed at the Mathematical Centre in Amsterdam, forerunner of the CWI. They were designed for Electrologica hardware, the EL X1 from 1958 and the EL X8 from 1965.

The X1 did not support ALGOL 60 implementation at all. Its ALGOL 60 system was designed by Dijkstra and Zonneveld and completed in 1960. Although developed as an academic exercise it soon was heavily used and appreciated.

The X8, successor of the X1 and (almost) upwards compatible to it, had a number of extensions chosen specifically to support ALGOL 60 implementation. The ALGOL 60 system, developed by Nederkoorn and Kruseman Aretz, was completed even before the first delivery of an X8.

In this document we describe the two systems, demonstrating the progress in both hardware and software in a relatively short period.

2000 Mathematical Subject Classification

01-08, 68-03, 68N20

1988 Computer science Classification

K.2, D.3.4

Keywords and Phrases

historical, ALGOL 60 compiler, Electrologica X1, Electrologica X8

Preface

In this document we compare two ALGOL 60 implementations, one for the Electrologica X1, completed in 1960, the other for the Electrologica X8, completed in 1965.

To give the flavour of the difference, we present below the results of some measurements, carried out with the help of emulations of the two machines, for one and the same program, a Runge-Kutta integration program applied to the calculation of planetary orbits.

		X1	X8
compiling	instructions executed	1 145 895	2 270 261
	compile time	61.9 sec	8.92 sec
	average instruction time	54.1 μ sec	3.93 μ sec
execution	instructions executed	9 450 329	1 705 131
	execution time	557.3 sec	10.36 sec
	average instruction time	52.6 μ sec	6.08 μ sec

Where the El X8 is about 12 times faster than the El X1, we see that program execution is more than 50 times faster. This is due to the fact that the X1 was developed for administrative applications in the first place, whereas the (upwards compatible) instruction set of the X8 was chosen with ALGOL-like languages in mind.

In this report we compare the two machines, the two compilers and the object codes they generate, in order to explain the figures given above, as well as many others. Moreover go we in somewhat deeper in aspects of the ALGOL 60 implementation of the X8 that have not been described elsewhere.

Contents

1	Introduction	1
1.1	The Mathematical Centre	1
1.2	Electrologica	2
1.3	ALGOL 60	3
1.4	ALGOL 60 implementation on the MC	4
1.5	Installation of the system	6
1.6	Maintenance of the ALGOL system for the X8	6
1.7	The Z8 committee	7
2	The Electrologica X1	9
3	The ALGOL 60 system for the EL X1	15
3.1	Introduction	15
3.2	The representation of values	17
3.3	The structure of the object program	18
3.3.1	Simple assignment statements	18
3.3.2	Array access	20
3.3.3	The <for-statement>	21
3.3.4	Procedure calls	25
3.3.5	Some standard functions	29
3.3.6	Designational expressions and goto statements	30

3.4	The structure of the compiler	31
3.4.1	The lexical scanner	32
3.4.2	The prescan program	32
3.4.3	The second or main scan program	32
4	The Electrologica X8	39
4.1	Floating-point instructions	40
4.2	New addressing variants	42
4.3	A new subroutine call	44
4.4	The execute instructions	44
4.5	Some other additions to the X8 instruction set	44
4.6	The execution times of floating-point operations	45
5	The ALGOL 60 system for the EL X8	49
5.1	Introduction	49
5.2	The structure of the object program	52
5.2.1	Simple assignment statements	52
5.2.2	Array access	53
5.2.3	The <for-statement>	56
5.2.4	Procedure calls	59
5.2.5	Some standard functions	62
5.2.6	Designational expressions and goto statements	64
5.2.7	Line numbers	66
5.2.8	Guarding stack overflow	67
5.3	The structure of the compiler	69
5.3.1	Overview	69
5.3.2	The General-purpose procedures	71
5.3.3	The translator scan	72
5.3.4	The macro processor	77

5.3.5	The symbol table	78
5.3.6	Prescan1	80
5.3.7	Prescan0	81
5.3.8	Results of some measurements	84
6	Comparison of the two ALGOL 60 systems	87
6.1	Comparison of the EL X1 and the EL X8	87
6.2	Comparison of the ALGOL 60 implementations	89
6.2.1	Comparison of the object programs	89
6.2.2	Comparison of the compilers	90
6.3	Results of measurements	92
6.3.1	Summary of results of Chapter 3 and Chapter 5	93
6.3.2	An estimation of the X1 to X8 acceleration factor	94
6.3.3	The programs	96
6.3.4	Compiling	97
6.3.5	Program execution	98
6.4	The influence of line-number book-keeping	100
7	Final remarks	103
A	Sample programs	105
A.1	Havie integrator	108
A.2	QR eigenvalues	112
A.3	JAZZ164, Runge-Kutta integration	118
A.4	Sieve of Erathostenes	125
A.5	Pentomino	128
A.6	Lisp interpreter	134
B	User-manual extracts	155

B.1	Extract of the X1 ALGOL user manual	155
B.2	Extract of the X8 ALGOL user manual	161
References		169

Chapter 1

Introduction

1.1 The Mathematical Centre

The Mathematical Centre (or shortly: the MC) was founded in 1946 in order to promote the application of mathematics in the Netherlands. The ideas for such an institute were developed during world war II and many parties, among which ZWO (Foundation for Pure Scientific Research), the city of Amsterdam, CBS (Central Bureau for Statistics), Philips Gloeilampen Fabrieken (now Philips Electronics) and the Bataafse Petroleum Maatschappij, contributed to its foundation.

Soon it had four departements: for pure mathematics, applied mathematics (mainly applied analysis), statistics, and numerical computation. The latter department was from 1947 to 1973¹ headed by Adriaan van Wijngaarden (who was president of the board of directors of the institute from 1961 until his retirement in 1981). After a study tour to the UK and the US the decision was taken to build a first computer, based on relay technology. The ‘ARRA’ was developed by C.S. Scholten and B.J. Loopstra and completed in 1952, but it never functioned satisfactorily and was broken down shortly after its official inauguration.

The structure of the next MC computer, called ‘ARRA II’, was highly influenced by G.A. Blaauw, who joined the MC from 1952 until 1955 and learned the profession in the US with Howard Aiken. It was completed in 1955 and worked successfully. It had a drum store of 1024 words of 30 bits (revolution time 20 msec). The basic software for

¹The department for numerical computation was split into two departments Januari 1st 1973, each with a new head.

the ARRA II was written by E.W. Dijkstra, who entered the MC in 1952. A slightly improved version of the machine, called FERTA, was build for Fokker, the Dutch aircraft factory.

The third computer of the MC was the ARMAC, again developed by Scholten and Loopstra. Its main store was again on drum, 3584 words (112 ‘pages’ of 32 words) of 34 bits, revolution time 13.3 msec. But in addition it had 2 pages of 32 words core store, with cycle time of 20 μ sec. One of these pages was free addressable and used as working area, the other contained permanently a copy of the drum page to which the instruction counter pointed (this copy was automatically loaded from drum whenever the instruction counter changed page). This meant that it was efficient to program loops in such a way that all its instructions were contained in one and the same page and its data resided in the free-addressable core page as much as possible. The ARMAC was set to work in 1956.

1.2 Electrologica

For the design of MC’s next machine, for which the Dutch assurance company Nillmij showed great interest, a company, Electrologica, was founded in 1956, with capital from the Nillmij and the know-how of the MC: some 45 people moved from MC to Electrologica. The first El X1 was delivered to the Nillmij in 1958. The Mathematical Centre got its copy in 1960. The X1 was fully transistorized, had core store (in units of 4096 words of 27 bits), cycle time 32 μ sec, an index register and an interrupt system for I/O. More details are given in Chapter 2. The basic software of the X1, including interrupt handling and the assembler, was again designed by Dijkstra. A full description of the X1 and of its basic software is given in Dijkstra’s thesis, defended in 1959 ([5]). The first ALGOL 60 implementation in the world that was operational was designed by Dijkstra and J.A. Zonneveld ([8]).

As a successor to the El X1, Electrologica designed the El X8. The first copy was (months too late) delivered in 1965 to the University of Utrecht, the MC got its copy in 1966. The development of software for the X8 was partly contracted out to the initial purchasers: the assembler to the Dr. Neher Laboratory of PTT, the Dutch post and telephone company, the ALGOL 60 implementation to the MC, a Fortran implementation to the Technical University of Kiel (Germany), whereas the Technical University of Eindhoven undertook the development of a multi-programming system. Most of these parties were represented in the Z8 committee (Z8, ‘zacht’, meaning soft, for software).

The El X8 was to a certain degree upwards compatible with the X1. The extension of the

instruction set was directed to the implementation of ALGOL 60; it contained floating-point hardware, stack instructions, an addressing mode fit for the access of local variables in a block structure, and an execute instruction for a fast parameter mechanism. It had core store (in units of 16384 words of 27 bits, cycle time 2.5 μ sec); a drum store (of 512 tracks of 1024 words) could be ordered as backing store.

Another aspect in which the X8 differed from the X1 was the treatment of I/O. In the X8 input/output was handled by a separate processor, 'Charon', on the basis of cycle stealing of the memory. For details on the X8 we refer to Chapter 4.

1.3 ALGOL 60

ALGOL 60 (ALGOL is an abbreviation of ALGORithmic Language) was the result of the work of an ad-hoc international committee. This committee consisted of famous scientists like Backus (yes, the founder of BNF), Bauer, John McCarthy (LISP), Naur, Rutishauser, Samelson, Van Wijngaarden, and Woodger. The final report was edited by Peter Naur.

The language was developed in the period from 1957 to 1960. It had ALGOL 58 as an early forerunner, in 1960 came the ALGOL 60 report (edited by Naur[1]), and in 1962 the final 'Revised Report on the programming language ALGOL 60' [18] was published.

ALGOL 60 was the first language whose grammar was defined in Backus Normal Form (BNF). It was of a great generality, thanks to the recursive definitions of BNF. It had (nested) block structure and an advanced parameter mechanism for procedures. It introduced the 'boolean' type as a full fledged citizen, with boolean variables, boolean constants, boolean expressions, and boolean procedures. Although it was designed in the first place as a language to express and communicate numerical algorithms, it lent itself also to express non-numerical algorithms in.

It had much success in Europe, where ALGOL 60 implementations were developed for numerous (commercial) computers, among which KDF9, TR4, Zebra, EL X1, and EL X8. Also some American companies developed ALGOL 60 systems: Burroughs, Honeywell. The success in the US was, however, hampered by the fact that IBM refused to support ALGOL 60, and that the physicists already had too much invested in FORTRAN, which was, from a scientific view point, inferior to ALGOL 60.

An aspect of ALGOL 60 that is not present in most other higher-order programming languages is the use of mixed-mode arithmetic expressions. ALGOL 60 does not have separate expressions of type integer and of type real, and in arithmetic expressions operands

of both types may occur. There are, however, strict rules stating under what conditions the result of an arithmetic expression is exact, i.e. of type integer. For exponentiation, e.g., the type of ‘ $\langle \text{primary} \rangle \uparrow \langle \text{primary} \rangle$ ’ is integer provided that both operands have integer type and the exponent is non-negative.

In assigning a real result to a variable declared to be of type integer that result is rounded to an integral value. The same occurs at other places where an integral value is needed, e.g. for an index in a subscripted variable.

The aspect of mixed-mode arithmetic expressions has some consequences for its implementation.

1.4 ALGOL 60 implementation on the MC

Where the MC was, rather early, involved in the development of the language ALGOL 60, it was almost self-evident that the MC would try to implement it. The work started November 1959 and already in August 1960 the first ALGOL 60 programs were compiled and executed. A formidable achievement! The work was done by four people. The compiler was written by Dijkstra and Zonneveld and was about 2000 instructions short, whereas the organisational and arithmetic subroutines supporting program execution (the ‘complex’) were written by Miss Römgens and Miss Christen (another 2000 instructions). The system was written for the EL X1 and could work in a machine with only 4K words memory. In 1961 a library containing more input/output procedures and some numerical subroutines were added to the system and in the middle of 1962 about 70% of machine time of the X1 was allocated to the compilation and execution of ALGOL 60 programs.

When I joined the MC in September 1962 the original crew of the ALGOL 60 project for the X1 was dissolved. Soon the ‘maintenance’ of the system became one of my tasks. Autumn 1963 I completed a load-and-go version of the system, greatly reducing paper-tape handling.

In 1962 a second ALGOL 60 implementation was completed, developed by Nederkoorn and Van de Laarschot[16, 17]. In contrast with the Dijkstra-Zonneveld system the Nederkoorn-Van de Laarschot system did a complete syntax check of the ALGOL 60 program. It produced rather compact object programs for a machine with variable word length, which could be executed by an interpreter for that machine written in X1 code. Consequently program execution was much slower than that with the Dijkstra-Zonneveld system and soon the Nederkoorn-Van de Laarschot system was used for syntax check only.

In the course of the year 1963 the MC ordered an Electrologica X8, the successor of the El X1, and undertook the task to develop an ALGOL 60 implementation for it. In July 1963 a working group was formed consisting of Zonneveld, Van de Laarschot, Barning, Nederkoorn, and myself. Nederkoorn, Van de Laarschot, and I defined the mapping from the source text in ALGOL 60 to the object code for the EL X8 and coded the subroutines necessary for the support of object code execution (where Van de Laarschot soon dropped out, this work was mainly done by Nederkoorn and myself). The result of this work, completed in 1964, was documented by Nederkoorn and by Kruseman Aretz and Nederkoorn but these internal reports were never published officially.

Barning contributed to the project in two ways. First he developed an emulator for the X8, to be run on the X1. This work was completed in 1963[2]. This emulator made it possible for Nederkoorn and myself to test object code of hand-translated ALGOL 60 programs and the execution-support subroutines already in an early stage of the project.

Barning's second task was the development of subroutines for the standard numerical functions of ALGOL 60 (*sqrt*, *sin*, *cos*, *ln*, *exp*, *arctan*[3]). Also he was able to test these functions with the X8-code emulator.

The compiler for the X8 was written by me. I decided to use ALGOL 60 itself as design language, in order to arrive at a well-structured and clear design. After some experiments I discovered that it was possible to use recursive procedures to carry out syntax analysis[9], a method that later was baptized 'recursive descent'. Having available a running ALGOL 60 implementation on the X1, it was possible to test large portions of this version of the compiler for the X8 (the size of the program necessitated to apply the Nederkoorn-Van de Laarschot implementation for this work). Work started in 1964 and was completed in 1965. The coding of the compiler into ELAN, the assembly language of the X8, took only one or two months[11]. In 1973 also (an updated version of) the ALGOL 60-text of the compiler was published[12].

Since the ELAN assembler under construction by the Dutch PTT was not ready in time it was good luck that the University of Utrecht (i.e. Van der Meulen), not satisfied with the developments in PTT, constructed another ELAN assembler. It was that one that operated early enough for me to test the compiler with.

Also the Electrologica operating system was not ready in time. It promised to be such large that in an X8 installation with 16K words core store not enough space was left for the ALGOL 60 system, and, according to its specifications, it required an extensive conversation between operator and machine even for the smallest program. For these reasons the MC decided, at a very late stage, to build its own operating system to support the execution of ALGOL 60 programs only. It was developed by Mailloux and van Berckel

and called ‘PICO monitor’, for one of the design criteria was minimum size.

1.5 Installation of the system

After one or two short test sessions at Electrologica’s premises at Rijswijk the system was installed on the X8 of Utrecht University by Mailloux and me in about two days, short after the machine’s delivery. We went to Utrecht with our bunch of paper tapes November 9th and 10th, and left a working system. I remember our astonishment over the machine’s speed: we tried a first ALGOL 60 test program and after reading the paper tape the machine seemingly failed to do any computation and remained idle. At last we detected that computation was ready before we could even lift our eyes from the tape reader.

Thereafter we visited Utrecht several times at increasing intervals, to solve small problems: November 12th, November 15th, November 16th, November 24th, December 7th, December 9th, December 20th, December 22nd, Januari 10th, Januari 17th. Right from the beginning the instability of the X8 and the imperfect hardware test programs of Electrologica were the main problem.

The Mathematical Center got its X8 in May, 1966. In spite of the enormous gain in computing power the machine was in no time fully occupied during day hours, which made it necessary to improve its operation, i.e., improve the operating system PICO. Gradually it was replaced by the MICRO system (1967) and the MILLI system. The latter I started to develop at the Mathematical Center, but I could only complete it after my switch to Philips Research Eindhoven in 1969. Thereafter it was also installed at the Mathematical Center. In MILLI four streams of programs (with different demands on the system) were executed in parallel.

For external clients the price for use of the X8 was Fl. 1000 per hour (for the X1 it was ‘only’ Fl. 600 per hour). Therefore computation times were reported to the users in ‘milli-hours’.

Both at the Mathematical Center and at Philips the X8 was operated in closed shop.

1.6 Maintenance of the ALGOL system for the X8

After its installation in 1965 the ‘maintenance’ of the ALGOL system for the X8 started. We mention here several aspects.

- The repair of errors. The most important problem showed up in the first week of use: if an ALGOL problem had a corrupt declaration and block structure, it could happen that the second or third scan of the compiler derailed, leading to a breakdown of the system. This was easily repaired by introducing beside the normal error message a ‘deadly’ error message causing the compiler to discontinue the analysis at the end of the first scan.
- The inclusion of the line printer as (main) output device. The inclusion of some background store on the drum. The inclusion of a plotter. The inclusion of punch cards as input/output medium. The latter required that next to the Flexowriter representation of basic symbols (with underlined key words) another representation was introduced (with key words between apostrophes).
- The purchase of a third cabinet of core store, with addresses from 32K to 48K - 1 commanding small addressing changes in order to allow stack growth beyond the 32K border.
- The input/output system was extended to allow in addition to Flexowriter code also ASCII for the representation of programs and data.
- The development of a library of procedures that could be used without declaration. Partly they were written in machine code, mainly those that had to do with input/output peripherals. But a large part was written as ALGOL 60 procedures and precompiled by a special version of the compiler. Their relocatable object codes were stored on the drum and were added selectively to a compiled program.

1.7 The Z8 committee

The Z8 committee was founded in 1963 by Electrologica and was headed by Van Wijngaarden. Its first meeting took place on September 23rd, 1963. Members were Dijkstra (THE), Van der Poel (PTT), Van der Sluis (RUU), Van Wijngaarden (MC), Kruseman Aretz (MC), Scholten (El), Schmidt(El), Zwanenburg (El), Kolff (El), Seligmann (El), and Dek (Nillmij). Zwanenburg functioned as secretary. Most meetings were attended also by Van der Meulen (RUU). The committee convened about twice a month. In total the Z8 had, according to the minutes, 50 meetings. Last meeting was June 23rd, 1966.

Quite a number of the early meetings were devoted to the definition of the assembly language, later called ELAN. Many discussions had also the ‘El-coordinator’ (operating

system) as subject. Also details of the hardware (e.g. interrupts) and proposals for change thereof were treated. The ALGOL 60 system under development at the MC was dealt with only a few times, mainly to discuss its inbedding in a simple operating system. June 17th, 1965 I communicated the completion of the ALGOL 60 version of the compiler, November 25th I promised to write a manual for the ALGOL system, and December 16th I announced that the system, brought to life on the Utrecht X8, was completed, for the time being. January 6th, 1966 I reported that a certain ALGOL 60 program (solving the Van der Pol differential equation numerically) was compiled about 5 times faster than on the EL X1, whereas program execution was 67 as fast.

The Z8 committee also demanded the possibility to order a special token drum for the line printer. That token drum contained all the special tokens from the ALGOL 60 alphabet (such as \vee , \wedge , \neg and $_{10}$) and the tokens $|$ and $_$ for the printing of symbols like \leq and \neq and to underline the word delimiters.

Chapter 2

The Electrologica X1

We give here a short summary of the EL X1 instruction set only. We do so using the notation of ELAN, the assembly language for the X8.

The word length of the X1 was 27 bits. The arithmetic was 1-complement. Instructions had an operation code of 12 bits and an address part of 15 bits. The memory was in units of 4K words, with a maximum of 24K. Moreover the X1 had some read-only memory, addressed from 24K (to maximally 32767).

There were three main registers: A, S, and B. Both A and S had 27 bits. Register B, the index register, had 16 bits only. There were three one-bit registers, C, LS, and OF. C was the condition register (C = 0 meaning yes), LS contained the sign of the result at condition setting, and OF was the overflow indication. Condition setting was not automatic, but should be asked for explicitly by an instruction variant, using the P(ositive), Z(ero), or E(qual) sign. All instructions could be made dependent of the condition, using the Y(es) or N(o) variant.

In general, most instructions allow one or more of three variant types. Let 'operation' be (the notation of) some X1 instruction (with none variant applied). Then:

1. Condition following variants: U, Y, or N.

ELAN notation	effect
Y, operation	if C = 0 then execute operation, else skip
N, operation	if C = 1 then execute operation, else skip
U, operation	execute operation, but do not change the value of the register involved in the operation

In the U(ndisturbed) variant, strictly spoken not a condition following variant, the result of the operation is formed in the exit register U of the accumulator, but not copied to one of the registers. The variant is useful only in combination with one of the next variants.

2. Condition setting variants P, Z, or E.

ELAN notation	effect
operation, P	operation; if $U \geq +0$ then $C:= 0$ else $C:= 1$
operation, Z	operation; if $U = +0$ or $U = -0$ then $C:= 0$ else $C:= 1$
operation, E	operation; if (signbit of U) = LS then $C:= 0$ else $C:= 1$

Moreover, in any of these three variants, as a last action LS is set to the signbit of U. Here, as in the previous variant, U is the exit register of the accumulator.

3. Addressing variants ‘:’ and B.

These variants influence the role of the address part in the instruction.

The :-variant is meaningful only if without that variant the operand of the instruction is a word in store, at the location indicated by the address part. Using the :-variant then means: use the address part itself as the operand (resulting in an operand between 0 and 32767).

The B-variant adds the value of B to the address part of the instruction in order to find the location of the store operand¹. In this variant B is therefore used as index register.

For many instructions the 12 bits of the operation code can be divided into 6 bits operation code proper and $3 * 2$ bits for the variants. In the tables below we indicate the value of the operation code proper as ‘ocp’.

The main instructions for register A are as follows. Let ‘label’ be an identifier denoting some address (i.e. some number between 0 and 32767).

Then we have²:

¹for $B \geq +0$ this addition is carried out modulo 32768, so that $32767 + B$ results in the address $B - 1$. This facilitates the use of B as stack pointer.

²addition and subtraction in the sense of the 1-complement arithmetic for 27 bits.

ocp	ELAN notation	effect	variants
0	A + label	A:= A + store[label]	UYN :B PZE
1	A - label	A:= A - store[label]	UYN :B PZE
2	A = label	A:= store[label]	UYN :B PZE
3	A = - label	A:= - store[label]	UYN :B PZE
4	label + A	store[label]:= store[label] + A	UYN B PZE
5	label - A	store[label]:= store[label] - A	UYN B PZE
6	label = A	store[label]:= A	UYN B PZE
7	label = - A	store[label]:= - A	UYN B PZE
20	A '+' label	A:= A '+' store[label]	UYN :B PZE
21	A '+' - label	A:= A '+' - store[label]	UYN :B PZE
22	A '*' label	A:= A '*' store[label]	UYN :B PZE
23	A '*' - label	A:= A '*' - store[label]	UYN :B PZE

Herein denotes '+' bitwise addition ($1 + 1 = 0$) and '*' bitwise multiplication.

Execution of these instructions costs about two store cycli (i.e. $64 \mu\text{sec}$), but in the :-variant only one store cycle (for loading the instruction itself).

Some examples:

1. to set condition register C = 0 without changing the value of any register other than C and LS:
U, A = 0, Z³
2. to load A with the absolute value of the location in store that is one place beyond the location referred to by B:
A = M[B+1], P
N, A = - M[B+1]
3. to replace the least significant 15 bits of A by zero:
A '*' -32767

The same instructions are available for register S (with ocp 8, ..., 15, 28, ..., 31).

The instructions 0 to 7 are also available for register B (with ocp 32 to 39). While loading B from store the 27 bits are truncated to 16 bits, whereas in the operations 36 to 39 the value of B is expanded by 11 copies of the sign bit.

³In ELAN, M is a standard identifier meaning address 0. Moreover, address identifiers may be indexed with numbers. Thus, for $0 \leq n < 32768$, M[n] means address n. Finally, :M[n] may be abbreviated to n.

In multiplication and division both A and S are involved. Denoting by AS the 53 bit register consisting of the 27 bits of A followed by the least significant 26 bits of S⁴ we have:

ocp	ELAN notation	effect	variants
16	MULAS(label)	AS:= S * store[label] + A	YN :B PZE
17	MULAS(-label)	AS:= S * (- store[label]) + A	YN :B PZE
18	MULS(label)	AS:= S * store[label]	YN :B PZE
19	MULS(-label)	AS:= S * (- store[label])	YN :B PZE
24	DIVAS(label)	S:= AS / store[label]; A:= remainder	YN :B PZE
25	DIVAS(-label)	S:= AS / (- store[label]); A:= remainder	YN :B PZE
26	DIVA(label)	S:= 2 ²⁶ *A / store[label]; A:= remainder	YN :B PZE
27	DIVA(-label)	S:= 2 ²⁶ *A / (- store[label]); A:= remainder	YN :B PZE

These instructions are slow: they cost 500 μ sec.

For transfer of control there are a.o. the following instructions:

ocp	ELAN notation	effect	variants
40	JUMP(label)	T:= T + store[label]	UYN :B
41	JUMP(-label)	T:= T - store[label]	UYN :B
42	GOTO(label)	T:= store[label]	UYN :B
42	GOTOR(label)	restore T,C,LS,OF,... from store[label]	UYN B
46	SUB0(:label)	store[8]:= T,C,LS,OF,...; T:= label	UYN
...
46	SUB15(:label)	store[23]:= T,C,LS,OF,...; T:= label	UYN

Herein is T the instruction counter. Also in these instructions T is incremented by 1 before execution of the instruction.

The U-variant here has a special meaning: if OF = 1, i.e., if integer overflow during addition or subtraction occurred, OF is set to 0 and control is transferred; otherwise the instruction is skipped.

There are 16 subroutine calls: SUB0, SUB1, ..., SUB15, differing only in the place where the linkdata are stored: store[8], store[9], ..., store[23]. The linkdata consist of the (old,

⁴In multiplications, the resulting sign bit of S is a copy of the sign bit of A; in DIVAS the sign bit of S is neglected.

but incremented) value of the instruction counter and the values of a number of one-bit registers like C, LS, OF, and some more (e.g. the interrupt permit). For the return from a subroutine one can use the GOTO-instruction, that restores T from the link but none of the one-bit registers, or the GOTOR-instruction, that restores both T and all the one-bit registers that are saved in the link.

The availability of 16 different subroutine calls makes it possible to have a hierarchy of subroutines. MC-convention was to call subroutines that do not call other subroutines by means of SUB0, and to use SUB_{i+1} to call a subroutine that calls internally one or more subroutines by SUB_i (and, of course, no subroutines by SUB_j with $j > i$). This convention is, however, no solution to implement recursion.

SUB14 and SUB15 were, at least for the X1 of the MC, reserved for the treatment of interrupts.

There are also counting jumps, with ocp 44 and 45, but their description is not necessary for this report.

There are many more X1 instructions. An important group are the 16 shift operations. There are four different circuits: A, S, AS, and SA. The circuit AS has 53 bits (the sign bit of S is excluded from the shift) and plays a role in arithmetical shifts, the circuit SA has 54 bits. There are also 2 kinds of shifts: round shifts, in which the bits that are shifted out at one side are entered again at the other side, and shifts-out, where bits are lost at one side and copies of the sign bit are supplied at the other side. Both shift types can, finally, be carried out to the right and to the left. The number of shift places is contained in the address part of the instruction; in the B-variant the value of B is added to it. Shift instructions are slow: they cost $40 + 8 * n$ μ sec, where n is the number of shift places.

Furthermore there are instructions for input/output, instructions for allowing and preventing interrupts, instructions for inverting the value of a register or copying its contents to another register, and stop instructions (transferring the X1 from the running state into the stopped state). Again we need not give details here.

Altogether the X1 has an elegant instruction set, which was pleasant to program in.

Chapter 3

The ALGOL 60 system for the EL X1

3.1 Introduction

The ALGOL 60 system for the EL X1 was developed in less than a year, from November 1959 to August 1960. An enormous achievement, certainly if one takes into account the fact that in that period the language itself was still under development, the machine still had to be delivered to the Mathematical Center, that machine had a core store of 4096 words only, its structure was more fitted to administration than to scientific computation¹, and the crew had no previous experience with the implementation of higher programming languages. Fortunately, Van Wijngaarden, head of the Computing Department, was member of the international committee designing ALGOL 60.

The main designers were E.W. Dijkstra and J.A. Zonneveld. They wrote the compiler, whereas a collection of subroutines (called the ‘complex’) to support execution of compiled programs was written by Miss M.J.H. Römgens and Miss S.J. Christen. The compiler length was about 2000 instructions, leaving about 2000 words working space during compilation. It had to read the source program, punched on paper tape in Flexowriter code, twice; during the second scan the compiled object code was punched on paper tape. The complex was also about 2000 instructions long, leaving 2000 words for object program and working space during program execution. Object code loading was a complicated process. The loader was overwritten during program execution. In total took compilation and ex-

¹The X1 had no floating-point hardware, no stack instructions, and no advanced addressing variants.

ecution of an ALGOL 60 program the reading (and subsequently rewinding) of about 10 tapes: the compiler tape, twice the tape(s) containing the ALGOL 60 source program, the complex tape, the loader tape, the second part of the object code, the cross-reference tape (of the library), the first part of the object code, the library tape (containing a collection of input/output routines and some numerical procedures) and, maybe, one or more tapes with input data read by the program in execution.

Execution was slow. Nevertheless, the ease of programming in ALGOL 60 compared to programming in assembly language made it rather popular. Also program punching and editing was changed dramatically by the introduction of the FRIDEN Flexowriter, which for the first time produced, while typing, both paper tape and a legible print. Within two years more than 70% of machine time was spent by the ALGOL system.

The implementation of the language was rather complete: there were only few and harmless restrictions on the language. It included block structure with local variables, arrays with dynamically determined bounds, (recursive) procedures, name and value parameters. The greatest shortcoming of the system was the almost complete absence of syntax checks during compilation and of bound checks of arrays during program execution.

In May 1962 the core store of the X1 at the MC was extended from 4K to 12K words. This extended the space available during program execution for object program and work area to 10K words. Somewhat later I used the extra core store to change the ALGOL system into a load-and-go system, thereby greatly reducing tape handling. The system tape now contained compiler, run-time support routines, loader, cross-reference data and the most frequently used library routines. During the first scan of the ALGOL 60 source program the text was stored, thus avoiding tape reading during the second scan, and the object code was also stored, thus avoiding its punching during the second scan and its reading in the loading phase. The necessary modifications of the system concentrated at the periphery of the compiler and the loader: no deep changes of the compiler were required.

In the next sections we go into some details about the representation of values, the structure of the object code produced by the compiler, and on the structure of the compiler itself. We will meet these subjects again in Chapter 5, where we describe the ALGOL 60 implementation for the EL X8.

3.2 The representation of values

Values of type integer are represented by one X1 word of 27 bits. Their value has to be between $-67\,108\,863$ and $+67\,108\,863$. This holds for both simple variables, array elements, and intermediate results of expressions.

Values of type real have two different representations. In a variable x it is represented by a mantissa m (with $0.5 \leq m < 1$) of 40 bits and a binary exponent e (with $-2047 \leq e \leq +2047$) of 12 bits (then $x = m * 2^e$). This representation requires two words per variable². This holds for both simple variables and array elements. An intermediate real result of an expression on the stack is represented by a mantissa of 53 bits and a binary exponent of 27 bits, requiring three words. Consequently, while loading the value of a real variable to the stack a change of representation is necessary, whereas the opposite change of representation is required while storing the (real) result of an expression in a variable.

The range for real variables is from about -10^{600} to about -10^{-600} and from about 10^{-600} to about 10^{600} (the value 0 being replaced by about 10^{-600}).

This representation for reals is meaningful: stored values take two words of store only, important for long arrays, whereas intermediate results of arithmetic expressions have representations that are fit for arithmetic operations.

All elementary operations (addition, subtraction, multiplication, division) take place at the top of the stack. This means for real values that the operations are carried out in 52 bits precision, i.e. in more than 15 decimals. Stored result, however, have a precision of 40 bits or about 12 decimals only. In certain applications one can make use of the higher precision of intermediate results on the stack. To compute the error in the exponential function³, e.g., one computes the difference between ‘ $exp(x)$ ’ and the series expansion in x for it, using enough terms to make the truncation error below 10^{-15} .

Boolean values are represented as integer values, using 0 for **true** and 1 for **false**.

²There was also a version of the ALGOL 60 system for the X1 using only one word for real variables, at the cost of precision. It was, however, hardly used and never maintained.

³In the X1 implementation of the standard functions the precision is 12 decimals only.

3.3 The structure of the object program

3.3.1 Simple assignment statements

The structure of the object code, generated by the Dijkstra–Zonneveld ALGOL 60 compiler can be illustrated by the code for the following ALGOL 60 statement:

$$i := i + 1$$

where i is a variable of type integer declared in the outermost block, with 138 as the address of its location and 10900 as the location of the integer constant 1.

instruction	explanation
B = 138	B:= address of i
SUB1(:TIAS)	Take Integer Address Static
B = 138	B:= address of i
SUB1(:TIRS)	Take Integer Result Static
B = 10900	B:= address of 1
SUB2(:ADIS)	ADd Integer Static
SUB1(:ST)	STore

We see that the code mainly consists of subsequent calls of subroutines of the complex of subroutines supporting the execution, which get their parameters(s) (if any) in one of the registers. These subroutines can be seen as the operation codes for a hypothetical stack machine, whereas a parameter in one of the registers can be looked upon as the corresponding address part of that instruction. In this respect the object code is almost P-code, the pseudo code for such a machine. Here, however, there is no need to decode the operation code to arrive at the proper subroutine.

The stack cells of the hypothetic stack machine consist of 4 words, the last of which indicates whether the cell contains the address of an integer variable or array (1 word), the address of a real variable or array (1 word), an integer value(1 word), or a real value (2 words mantissa, 1 word binary exponent). The stack pointer of the stack machine is administrated in an X1 store location labeled ‘AW’ (Accumulator Wijzer).

As illustration of a subroutine of the complex we give the code for TIRS:

label	instruction	explanation
TIRS:	A = 4]
	AW + A] AW:= AW + 4
	A = M[B]	integer number to A
	B = AW	stack pointer to B
	M[32764 + B] = A	store integer in stack cell
	S = -0]
	M[32767 + B] = S] indication for integer value
	GOTOR(M[9])	return to object code

If variable i is local to an inner block, e.g. a procedure body, it has a two-component address, consisting of a block number bn and a displacement d , from which its location in store is derived dynamically (hence the term ‘dynamic address’). In that case the value of i is loaded to the stack by the instruction pair:

instruction	explanation
S = 32 * d + bn	S:= dynamic address of i
SUB1(:TIRD)	Take Integer Result Dynamic

Subroutine TIRD starts with a call ‘SUB0(:STAT)’, converting the dynamic address in register S into a static address in register B, and is continued by the code of TIRS⁴.

The fact that ALGOL 60 has mixed mode arithmetic expressions, combined with the fact that the representation of integer and real values differ in the ALGOL 60 implementation for the X1, has many consequences. So all (dyadic) arithmetic operations must inspect whether both operands have the same type. If not, the operand with integer type is converted to real representation first. Also the subroutine ST (i.e. STore), finding an address and a value on top of stack, has to inspect the two types and, when necessary, to convert an integer value to real representation or to round a real value to an integer.

In total execution of the ALGOL 60 statement ‘ $i := i + 1$ ’ costs:

addressing mode	instructions	time
static	60	3284 μ sec
dynamic	74	4044 μ sec

⁴Hence, following the X1 conventions for subroutine nesting, both TIRD and TIRS are called by SUB1 with its link in M[9].

Let now x and y be variables of type real, declared in the outermost block. Then the translation of statement:

$$x := x + y$$

reads analogously:

instruction	explanation
B = @ x	B:= address of x
SUB1(:TRAS)	Take Real Address Static
B = @ x	B:= address of x
SUB1(:TRRS)	Take Real Result Static
B = @ y	B:= address of y
SUB2(:ADRS)	ADd Real Static
SUB1(:ST)	STore

where we denoted the address of x by @ x and that of y by @ y . The number of instructions executed in subroutine ADRS depends on the values of x and y , and so does execution time. For initial values for x of 3.14 and for y of 0.1 we find⁵:

instructions	time
110	5784 μ sec

Statement $x := x \times y$ is executed in 6676 μ sec by 101 instructions, statement $x := x/y$ in 6946 μ sec by 105 instructions.

3.3.2 Array access

Our second example deals with array access. Consider the following ALGOL 60 statement:

$$R[i] := R[i] + x$$

Let R be a real array, x a real variable, and i a variable of type integer, and all three be declared in the outermost block (and therefore addressed statically). Let us denote the

⁵For values 0.1 of x and 3.14 of y 114 instructions are executed, lasting 5980 μ sec.

address of the storage function⁶ of R by $@R$, the address of x by $@x$ and the address of i by $@i$. Then we have the following object code for that statement:

instruction	explanation
B = @R	B:= address of storage function of R
SUB1(:TRAS)	Take Real Address Static
B = @i	B:= address of i
SUB1(:TIRS)	Take Integer Result Static
SUB1(:IND)	INDEXer
B = @R	B:= address of storage function of R
SUB1(:TRAS)	Take Real Address Static
B = @i	B:= address of i
SUB1(:TIRS)	Take Integer Result Static
SUB1(:IND)	INDEXer
SUB1(:TAR)	Tranform Address into Result
B = @x	B:= address of x
SUB2(:ADRS)	ADD Real Static
SUB1(:ST)	STore

The number of instructions that are executed for the above piece of object code and the execution time will again depend on the values of $R[i]$ and x . For the values 0.1 and 3.14 for $R[i]$ and x , respectively, 216 instructions are executed in 11 532 μsec .

Due to the fact that boolean array elements are stored one element per word, the code for the statement ' $B[i]:= \text{true}$ ' is the same as for ' $I[i]:= 0$ ' (where B is a boolean array and I an array of type integer): both 8 instructions, demanding the execution of 81 instructions in 4 416 μsec .

3.3.3 The <for-statement>

The structure of the ALGOL 60 <for-statement> is complex, leading to complex object code. Consider the following ALGOL 60 statement:

```
for d:= 2, 3, 5 step s until q do
  begin q:= n ÷ d; s:= 6 - s;
```

⁶For the declaration of an array of dimension n a set of $n+3$ consecutive words is constructed in store containing information about the total number of elements of the array, the location of the elements in store, and whether one word (for type integer and boolean) or two words (for type real) are assigned per array element.

```

    if  $q \times d = n$  then goto aa
end

```

We see that the <for-list> can contain several <for-list-element>s of different type. For all these elements the same controlled statement has to be executed zero or more times. Therefore an additional anonymous variable is needed to record which is the current element. We see also that both the step size and the upper bound in an step-until-element can be changed by the controlled statement.

The semantics of the step-until-element 'A **step** B **until** C' is described in the Revised Report as:

```

    V:= A;
L1: if  $(V - C) \times \text{sign}(B) > 0$  then goto Element exhausted;
    Statement S;
    V:= V + B;
    goto L1;

```

In each iteration step the values of V and B are needed twice, whereas assignments to V are done both initially and in each next iteration step. This makes implementation complicated.

In the ALGOL 60 implementation for the X1 each for-statement acts as a parameterless procedure. This creates the possibility to have new local variables (and has the additional advantage to prevent jumps from outside the for-statement into it, by the same mechanism that prevent jumps into procedure bodies). But the extra block administration has a price in execution time.

In fact, three additional variables are used: one regular local variable of the procedure (created in subroutine FOR0, see below), the link in the link data of the procedure, and a word that is reserved in the stack before the creation of the block administration of a procedure call⁷.

The local variable is used to store the address of the controlled statement (i.e. L4, see below). The procedure link serves two purposes: to save the link of the call of subroutine FOR5 and, in subroutine FOR1, to proceed the elaboration of the step-until element after the first iteration step. The third word mentioned above is used to record the value of 'sign(B) + 8' and influences the execution of both subroutine FOR1 and subroutine FOR7.

⁷This word is filled by a zero in case of a procedure statement and by the address of the function result in case of a function designator.

The object code for statement

for i:= 1 step 1 until n do S

reads:

label	instruction	comment
	GOTO(:L2)	" jump over code for address of <i>i</i>
L1:	B = 138	" B:= address of <i>i</i>
	SUB1(:TIAS)	" Take Integer Address Static
	SUB2(:FOR1)	
	GOTO(:L4)	" jump to code for statement S
L2:	A = 0	" no parameters
	B = :L3	" address of the 'procedure body'
	SUB1(:ETMP)	" call the 'procedure'
	B = 10900	" address of constant 1
	SUB1(:TIRS)	" Take Integer Result Static
	SUB0(:FOR5)	" initial value
	B = 10900	" address of constant 1
	SUB1(:TIRS)	" Take Integer Result Static
	SUB3(:FOR6)	" step size
	B = 139	" address of <i>n</i>
	SUB1(:TIRS)	" Take Integer Result Static
	SUB3(:FOR7)	" upper bound
	S = :L5	" address of code following the for-statement
	GOTO(:FOR8)	" exit for-statement
L3:	SUB0(:FOR0)	" block entrance
	GOTO(:L1)	" jump to code generating address of <i>i</i>
L4:	...	" object code for statement S
	GOTO(:L1)	" jump to code generating address of <i>i</i>
L5:	...	" code for next statements

Its explication is a disaster, but I will give it a try.

First its static structure. It consists of the following parts:

1. a jump over Part 2 to Part 3;
2. code to load the address of the controlled variable to the stack, followed by a call of FOR1 and a jump over Parts 3, 4, 5 and 6 to Part 7, the controlled statement S;

3. three instructions to do the block entrance
4. code for the for-list-elements. Here we have just one element, a step-until element. Its code consists of three component, for loading the initial value, the step size, and the upper bound, terminated by calls of FOR5, FOR6 and FOR7, respectively;
5. two instructions for exiting the for-statement;
6. an instruction initializing the loop, followed by a jump to Part 2;
7. object code for statement S;
8. a jump to Part 2.

Next the dynamics.

Part 1 leads immediately to Part 3. This is the standard introduction to a block, coded as a parameterless procedure (hence $A = 0$) with its body starting at location L3 (hence $B = :L3$). ETMP puts a zero on top of stack and next constructs on the stack a new block cell, containing all the necessary link data. The link of the call of ETMP is saved there. In normal procedure calls it points to the code where the program is continued after completion of the execution of the procedure. Here it points to Part 4, the for-list-element(s).

Returning from ETMP, Part 6 completes the block introduction. It reserves space for one local variable. Execution is transferred to Part 2.

Here first the address of the controlled variable is loaded to the stack. Next FOR1 is called. It saves its link in the local variable of the block. Since it finds a zero below the block cell it jumps via the ETMP link in the link data to Part 4.

Part 4 loads the initial value to the stack and calls FOR5. FOR5 changes the link of ETMP in block's link data to its own link (consequently, the preceding code of PART 4 will not be executed another time!) and returns. Now the test $(V - C) * \text{sign}(B)$ has to be performed. Therefore, in Part 4 now the step size is loaded, followed by a call of FOR6, in which, since it finds a zero below the block cell, first the initial value is assigned to the controlled variable, next the value of the step size on top of the stack is replaced by its sign and, incremented by 8 (therefore different from 0) written below the block cell in order to change the future behaviour of FOR1 and FOR6. Next the upper bound is loaded and FOR7 is called. In FOR7 the test is carried out. If the element is not yet exhausted, the cells with the address of V, the value of V and the upper bound are removed from the stack and control is transferred via the local variable, i.e. the saved link of FOR1, to the

last instruction of Part 2, i.e. a jump to Part 7. If, however, the element is exhausted, the link of FOR 7 is stored in link data of the block, a zero is written back just below the block cell (giving FOR1 and FOR6 their original meaning) and control is transferred via FOR 7's link to the instructions following the call of FOR7, i.e. either to the next for-list-element or, in case the for-list is exhausted, to Part 5, the exit procedure of the for-statement.

Part 7 is the execution of statement S. It is concluded by a jump to Part 2.

Now we get a repetition of previous steps, with modified effect of FOR1 and FOR7. In FOR1 now the value of V is loaded to the stack (using the address on top of the stack), and control is transferred via the link in the link data of the block, i.e. to the instruction following the call of FOR5 in PART 4. In Part 4 the step size is loaded. In the call of FOR6 it is now added to the value of V before storing the result in V.

Complicated, isn't it? Definitely the ideas of 'structured programming' had not been invented yet. Especially the change of effect of subroutines FOR1 and FOR7 is nasty, as is the fact that some of those subroutines do not end by returning via their link but pass control to some other instructions of the object program. Even with the compiling technique used to generate the object code something simpler and more structured could have been obtained.

No wonder that this complicated construction also takes time. Taking for S the <dummy statement>, with an object code of 0 instructions, the execution of statement

for i:= 1 step 1 until n do S

takes:

for $n > 0$: $10\,044 + n \times 7\,612 \mu\text{sec}$,
for $n \leq 0$: $10\,772 \mu\text{sec}$.

3.3.4 Procedure calls

Let procedure p be declared in the outermost block by:

```
procedure p(z); real z;  
begin real y;  
end;
```

and let x be a simple global variable of type real with initial value 3.14.

Then the call p(x) generates the following object code:

label	instruction	comment
	B = @p	" load address declaration of p in register B
	GOTO(:L1)	" jump over parameter code
	(0 + @x)	" parameter code word for x
L1:	A = 1	" number of actual parameters in register A
	SUB1(:ETMP)	" ExTransMark Procedure

with execution time 4540 μ sec, of which 1560 μ sec for the execution of the body, and 2980 for the call administration and for parameter conversion (see below),

whereas the code for $p(x+0.1)$ reads:

label	instruction	comment
	B = @p	" load address declaration of p in register B
	GOTO(:L1)	" jump over parameter code
L0:	B = @x	" B:= address of x
	SUB1(:TRRS)	" Take Real Result Static
	B = @0.1	" B:= address of constant 0.1
	SUB2(:ADIS)	" ADd Integer Static
	GOTO(:EIS)	" End of Implicit Subroutine
	(2^{20} + :L0)	" parameter code word for $x + 0.1$
L1:	A = 1	" number of actual parameters in register A
	SUB1(:ETMP)	" ExTransMark Procedure

with the same execution time.

Now let us change the context to procedure $p1$ (again declared in the outermost block):

```

procedure p1(z); real z;
begin real y;
    y:= z
end;

```

Now the execution time of $p1(x)$ is 8912 μ sec, of which 4372 μ sec for the assignment $y:= z$, whereas the execution time of $p1(x+1)$ increases from 4540 μ sec to no less than 17824 μ sec, of which 13284 μ sec for the assignment $y:= z$.

For an explanation of these figures we need to go into some detail about the addressing mechanism. For each block that during program execution is entered but not yet exited there is a block cell in the stack. A block cell contains some link data, parameter data, local data and, perhaps, some stacked intermediate results. The link data consist of:

the procedure link proper (i.e. the link of the call of ETMP), the block number bn , the dynamic link (i.e. the address of the block cell which was active before the current block was entered), and the static link (i.e. the address of the block cell belonging to the most recent incarnation of the textual enclosing block). Moreover the block cell contains the value of the stack pointer between statements, i.e. of the stack without any stacked intermediate results.

In order to be able to convert a two-level address (bn, d) (where bn is the block number and d is the displacement with respect to the begin of the block cell) to a static address n , a display $disp$ is maintained: $n = disp[bn] + d$. The elements of $disp$ are the begin addresses of the block cells in the so-called static chain: $disp[0] = 0$, $disp[BN] = PP$ (with BN being the block number of the most recently activated block and PP the begin address of its block cell), and for $0 < i < BN$ we have $disp[i] =$ the static link from the block cell starting at $disp[i + 1]$. This display is located at a fixed place, where 32 words are reserved for it.

Consequently it is necessary to update this display at every change of context: at block entry (here ETMP does the main job), at block exit (either by passing the block's end, where RET, i.e. RETurn, does the work), or by a goto statement (where GTA, i.e. GoTo-Adjustment, carries out the display update, cf. Section 3.3.6.). Moreover, when in the procedure body one of the formal name parameters is accessed, a temporary change of context is necessary from the procedure body's context to the context of the corresponding actual parameter, i.e. to the context of the procedure's call. This implies two display updates. As an optimization, however, for an actual parameter which is either an identifier or a constant no such updates are necessary.

For each actual parameter that is more complicated than just an identifier or a constant a subroutine (called 'implicit subroutine' or 'isr') is constructed, preceding the call (cf. the code for the call of $p(x + 0.1)$). Following the implicit subroutines there is for each actual parameter a one-word actual parameter descriptor, containing information about the address and the nature of the parameter. In turn these are followed by an instruction that puts the number of actual parameters in register A, before ETMP is activated.

The nature of the actual parameter is described by $2 * 2$ bits. Two of these are used by ETMP, the other two are used by the complex routines TFA and TFR (see below).

ETMP constructs the new block cell on top of the stack. It fills in the link data and constructs for each actual parameter a two-word parameter descriptor on the basis of its one-word descriptor preceding the call of ETMP. The second word of the parameter descriptor in the new block cell contains both block number and address of the old block cell (the calling context). The first word contains the two bits information about the

nature of the actual parameter to be interpreted by TFA and TFR⁸ plus an address. For an identifier or an constant it is always its static address: dynamic addresses are elaborated by ETMP using the old context. It can also be the address of the corresponding implicit subroutine.

After the construction of these two-word parameter descriptors ETMP transfers control to the translation of the procedure declaration.

The translation of the procedure declaration starts with two instructions:

B = bn	” B:= the procedure’s block number
SUB0(:SCC)	” Short CirCuit

by which the address of the new block cell is added to the display. Next follows code to evaluate value parameters, to handle the declarations of local variables and the code for the statements of the procedure body. As an example we give the code for the statement ‘y:= z’ from procedure *p1*.

S = @y	” S:= dynamic address of y
SUB1(:TRAD)	” Take Real Address Dynamic
S = @z	” S:= dynamic address of z
SUB1(:TFR)	” Take Formal Result
SUB1(:ST)	” STore

TFR analyses the nature of the actual parameter from its two-word descriptor. If the actual parameter is a simple variable or a constant, it can immediately load its value, without any change of context. This is the case in the call ‘p1(x)’, and then execution of TFR takes 1992 μsec . If, however, there is an implicit subroutine for the actual parameter, TFR has to carry out the context switch to the calling environment. This is the case in the call ‘p1(x+0.1)’, and this time TFR alone takes 4424 μsec , the evaluation of ‘x+0.1’ 3848 μsec , and the switch back, carried out by EIS, reinstalling the context of the procedure body, 2796 μsec . Hence the loading of the parameter value ‘x+0.1’ has a total cost of 11068 μsec .

The translation of a statement ‘z:= z + 0.1’ (only allowed for calls with a variable as actual parameter) reads:

⁸Thereby discriminating the following four situations: a simple variable or constant of type real, a simple variable or constant of type integer or Boolean, an implicit subroutine delivering an address, or an implicit subroutine delivering a value.

S = @z	” S:= dynamic address of z
SUB1(:TFA)	” Take Formal Address
S = @z	” S:= dynamic address of z
SUB1(:TFR)	” Take Formal Result
B = @0.1	” B:= @0.1
SUB1(:ADRS)	” ADd Real Static
SUB1(:ST)	” STore

which, for initial value 3.14 of z , leads to an execution time of 7936 μsec for 152 instructions executed.

The code for the declaration of a procedure ends with the instruction ‘GOTO(:RET)’, RETurn. Routine RET of the complex removes the procedure’s block cell from the stack and carries out the context switch to the calling environment by updating the display using the static chain in the link data of block cells. The jump to RET and RET’s execution together take here 852 μsec .

3.3.5 Some standard functions

The standard functions *sqrt*, *sin*, *cos*, *ln*, and *exp* were implemented as subroutines of the complex, transforming the top-of-stack value. There were no provisions for using these functions as actual parameter of a procedure. Function *arctan* was originally also a subroutine of the complex, but later replaced by an MCP, i.e., one of the procedures of the library. The object code for statement

$$x := \sin(y)$$

reads:

B = @x	” B:= static address of x
SUB1(:TRAS)	” Take Real Address Static
B = @y	” B:= static address of y
SUB1(:TRRS)	” Take Real Result Static
SUB0(:SIN)	” call of sine routine
SUB1(:ST)	” STore

Function *sqrt* is computed by improving an initial estimation by three steps in Newton’s iteration scheme. $\cos(x)$ is computed as $\sin(x + \pi/2)$. Functions *sin*, *ln*, and *exp* are

computed by reducing their arguments to some narrow intervals and using polynomial approximations.

Below we give some data on the execution performance. They were measured as the average for a range of values. For *sqrt*, *exp*, *ln*, and *arctan* we used the arguments 1.0 (1.0) 20.0, for *sin* and *cos* the values $-\pi$ ($\pi/10$) $+\pi$.

function	instructions	time (μ sec)
<i>sqrt</i>	47	4 453
<i>sin</i>	229	24 557
<i>cos</i>	298	28077
<i>exp</i>	232	24 195
<i>ln</i>	369	28 010
<i>arctan</i>	1500	106 523

3.3.6 Designational expressions and goto statements

The object code for a designational expression always leads to a transfer of control to the selected label; therefore the compiler just ignores keyword '**goto**'.

The object code for statement:

goto if b then aa else bb

is just the same as for statement:

if b then goto aa else goto bb

and reads (for local labels *aa* and *bb*):

	B = @b	" load address of variable b in register B
	SUB1(:TIRS)	" Take Integer Result Static
	SUB0(:CAC)	" Copy Boolean Accumulator into Condition
N,	GOTO(:L0)	" conditional jump to else-part
	GOTO(@aa)	" jump to label aa
	GOTO(:L1)	" jump over else-part
L0:	GOTO(@bb)	" jump to label bb
L1:		

Statement:

goto aa

is, for non-local label *aa*, translated as:

B = n	" load blocknumber n of label aa in register B
SUB0(:GTA)	" GoTo-Adjustment
GOTO(@aa)	" jump to label aa

Statement:

p(aa)

with *aa* some (nonformal) label and *p* some (nonformal) procedure, is translated as:

B = @p	" load adress declaration of p in register B
GOTO(:L1)	" jump over parameter code
L0: B = n	" load blocknumber n of label aa in register B
SUB0(:GTA)	" GoTo-Adjustment
GOTO(@aa)	" jump to label aa
GOTO(:EIS)	" End of Implicit Subroutine
($2^{20} +$:L0)	" parameter code word for aa
L1: A = 1	" number of actual parameters in register A
SUB1(:ETMP)	" ExTransMark Procedure

Evaluation of the corresponding formal parameter leads already to transfer of control to label *aa*.

The use of labels as value parameter of a procedure was prohibited. If done it lead to curious and unpredictable derailments.

3.4 The structure of the compiler

The ALGOL 60 compiler for the X1 consists of 70 pieces of code with length varying from 1 to 170 instructions, placed in some random order. Some pieces are subroutines, others end by a jump to some other piece.

It is, however, possible to discern some components: the lexical scanner, the prescan program and the second-scan program.

3.4.1 The lexical scanner

There are 4 subroutines, which, together, build kind of a lexical scanner. The lowest level thereof reads tape symbols, the uppermost level delivers delimiters in some internal representation. The latter is named ‘read–until–next–delimiter’. If it encounters an identifier or a number between two delimiters, that identifier or number is assembled and the result is delivered at a fixed place; moreover, a boolean value indicates whether that was the case. In total 529 instructions⁹.

3.4.2 The prescan program

There are three pieces that together constitute the first–scan program of the compiler. Its task is to make a list, with one element for each block, procedure declaration or for statement of the program. Each of these elements contains in turn two sublists, one containing the names of all procedures declared locally in the block or procedure, the other with the names of all labels or all switches occurring or declared locally¹⁰. The names of procedures, labels, and switches are listed *without* any descriptor at all: just the identifiers and nothing more. The total length of these three pieces of code is 173 instructions only; it is hardly imaginable that it is possible to interpret an ALGOL text to such a degree as is necessary to isolate the names of certain specific objects with so few instructions. The prescan part can be characterized as the art of intelligent text skipping!

3.4.3 The second or main scan program

The remaining pieces constitute the second scan program. It contains a main cycle consisting roughly of the following actions:

1. read until next delimiter,
2. if thereby a number or an identifier has been encountered, look it up,
3. jump to the piece of code belonging to the delimiter just read.

As an example we give the code belonging to delimiter **step** (transcribed from X1 notation into ELAN):

⁹This figure holds for the load–and–go version; the original version is probably not much different.

¹⁰In contrast to the ALGOL 60 report, the ALGOL 60 implementation for the X1 prescribes a certain order for the declarations of a block or procedure: first scalar variables, next the arrays, and finally switches and procedures. Therefore, in the second text scan, when the object code is produced, at all applied occurrences of scalars and arrays their declaration has been passed already. This is not true for applied occurrences of labels, procedures, and switches.

step:	SUB2(:ETT)	” Empty stack Through Thenelse
	A = 24	” A:= object-code number of FOR5
	S = 0	” S:= 0
	SUB0(:FRL)	” Fill Result List(FOR5,0)
	GOTO(:MAIN)	” return to main cycle

Subroutine FRL has two parameters; often the second parameter is superfluous, as is the case here. It just adds an instruction to the object code.

The task of ETT is to complete the translation of an expression. Thereafter the subroutine call of FOR5 is generated, which concludes the translation of the initial-value expression of a step-until element of a for-statement (c.f. Section 3.3.3).

Compiler subroutine ETT is called in the sections belonging to delimiters ‘do’, ‘,’’, ‘:’, ‘step’, ‘until’, ‘while’, and ‘end’, all belonging to the (in modern parsing terminology so-called) Follow Set of an expression. Its task is, as said before, to complete the translation of the expression. In the first place it can be the case that the last call of read-until-next-delimiter passed an identifier or a constant. Then still code for loading that operand has to be generated. Next it is possible that the stack contains some operators that were put there in the course of the transformation of the infix notation of the ALGOL 60 expression to the postfix notation of the object code. For all these operators code has to be generated. Finally the stack can contain a number of **else**-symbols, indicating that the expression is the else-part of one or more conditional expressions. In that case the destination address of one or more jump instructions over the else-parts have to be filled in¹¹.

The code of ETT reads:

ETT:	A = M[10]	
	W36 = A	” save link to working space 36
	A = 1	
	OFLA = A	” oflag:= true
ETT[4]:	SUB1(:POP)	” Production of Object Program(1)
	SUB1(:THENELSE)	” THENELSE?
Y,	GOTO(:ETT[4])	” if so, repeat last instructions
	GOTO(W36)	” return via saved link

¹¹In other sections belonging to a delimiter from the Follow Set of an expression there is no call of ETT but a cycle of 3 instructions calling POP and THENELSE directly.

The first two tasks mentioned above are carried out by the call of Production-of-Object-Program, which has an operator priority as parameter, priority 1 being the lowest priority in expressions. THENELSE is a Boolean function that fulfills the third task and delivers in condition register C whether it found an **else**-symbol in the stack.

The interpretation of delimiters in general depends on the context. A clear example is the comma, which can separate subscript expressions, for-list elements, actual parameters, etc. In the compiler the context is administrated by means of 6 Boolean variables:

name	context
<i>eflag</i>	an expression
<i>oflag</i>	the start of an expression
<i>mflag</i>	an actual parameter list
<i>iflag</i>	a subscript list
<i>vflag</i>	a for clause
<i>sflag</i>	a switch declaration

The code of compiler subroutine ETT given above sets ofla to true. In the case of the call of ETT in the code belonging to delimiter **step** this effectuates the context transition from ‘inside an (arithmetic) expression’ to ‘at the start of an (arithmetic) expression. It should be the case that, prior to that call of ETT, *eflag* = true, *oflag* = false, *mflag* = false, *iflag* = false, *vflag* = true, and *sflag* = false, but *this is not checked!* Here we see illustrated that the compiler expects the ALGOL 60 program to be correct.

As an example of the use of a context variable we present the code of the section belonging to delimiter ‘)’:

CLOSE:	A = MFLA, Z	” mflag = false?
	N, GOTO(:COMMA[32])	” else join with parameter separator
CLOSE[2]:	SUB1(:POP)	” Production of Object Program(1)
	SUB1(:THENELSE)	” THENELSE?
	Y, GOTO(:CLOSE[2])	” if so, repeat last instructions
	A = 1	
	TLSC – A	” stack pointer:= stack pointer – 1
	S = :MFLA	
	SUB0(:LTF)	” retrieve old value of mflag from the stack
	GOTO(:MAIN)	” return to main cycle

If *mflag* = true then delimiter ‘)’ is interpreted as the closing parenthesis of a procedure statement or a function designator. In that case control is transferred to part of the section

for delimiter ‘,’, where the code for the last actual parameter is completed and next the call of the procedure, preceded by actual parameter code words, is generated.

Otherwise, delimiter ‘)’ is interpreted as the closing parenthesis of an expression-between-parenthesis. The code for the expression is completed by an alternation of calls of sub-routines POP and THENELSE in the way we discussed above, the opening parenthesis is removed from the stack and the old value of *mflag* is retrieved from the stack. Note that the rest of the context remains unchanged: it is typical for this compiler that at context switches only that part of the context is saved in (and later retrieved from) the stack that might be different in the new context. Note also that it is assumed that there is an opening parenthesis on top of the stack and a saved *mflag* value underneath without any check whether that is indeed the case!

Another example of a context switch is given by the code of the section belonging to delimiter ‘[’:

SUB:	A = EFLA, Z	” eflag = false?
Y,	SUB2(:RLA)	” if so, reserve area for local arrays
	A = 1	
	OFLA = A	” oflag:= true
	A = 0	
	OH = A	” operator priority level:= 0
	S = EFLA	
	SUB0(:FTL)	” put eflag to the stack
	S = IFLA	
	SUB0(:FTL)	” put iflag to the stack
	S = MFLA	
	SUB0(:FTL)	” put mflag to the stack
	S = FFLA	
	SUB0(:FTL)	” put fflag to the stack
	S = JFLA	
	SUB0(:FTL)	” put jflag to the stack
	S = NID	
	SUB0(:FTL)	” put nid to the stack
	A = 1	
	EFLA = A	” eflag:= true
	IFLA = A	” iflag:= true
	A = 0	
	MFLA = A	” mflag:= false
	SUB0(:FTD)	” put $256 \times \text{oh} + '['$ on the stack
	S = JFLA, Z	” jflag = false?
Y,	SUB1(:GAI)	” if so, Generate Address of Identifier
	GOTO(:MAIN)	” return to main cycle

If *eflag* = false, it is possible that delimiter '[' opens the subscript list of a subscripted variable that is the left part of an assignment statement. In that case it can be the first statement of a block, thus marking the end of the declarations of the block. In that case perhaps still the areas of the local arrays have to be reserved, which is checked (and carried out) by subroutine RLA (Reserve Local Arrays). Next we see a context change, saving old values and setting new values to the context¹².

¹²We see two other flags here, *fflag* and *jflag*. They are derived from the identifier preceding the delimiter and indicate whether that identifier is formal and whether it is a switch identifier. Variable *nid* points to the descriptor of the identifier. We see here the only occasion where *nid* is saved to the stack:

It is striking how short the code of these examples is. It also clear that adding syntax checking to this compiler would be a formidable job.

As said before, the most complicated section is that for delimiter ‘,’. Its structure is the following¹³:

```

if iflag then deal with subscript separator else
if vflag then deal with for-list separator else
if mflag then deal with actual parameter separator else
begin ETT;
  if sflag then deal with switch declaration
  else deal with array declaration
end

```

Therefore that section is one of the longest: 145 instructions.

There was hardly any check on errors in the source program. Most checks had to do with the lexical level, e.g. a parity error in a paper tape punching. Further there were checks on store management. The only (context-sensitive) syntax check was regarding the applied occurrence of an undeclared identifier. At a later stage the occurrences of brackets were counted, and it was tested at each occurrence of a semicolon or an **end** symbol whether there were no unmatched square or round brackets. At the occasion of a failing test the X1 came to a full stop, showing an error number, without a possibility to continue: no back-on-the-rails procedure.

There was a library system of machine-code procedures and functions that could be used without any declaration in the source program. These were selectively loaded from paper tape at the end of the (object code) loading phase. That system contained mainly input-output procedures and some simple numerical subroutines.

We hope that these examples suffice to give a fair impression of the X1 ALGOL 60 compiler of Dijkstra and Zonneveld. Recently the full code of the compiler with much more explanation of details is documented in [8].

in general the code for loading the address of an identifier is generated immediately (here by GAI), but in the case of a switch designator that generation is postponed and carried out after the compilation of the subscript expression.

¹³Comma's occurring in procedure headings are dealt with in the section belonging to delimiter **procedure**.

Chapter 4

The Electrologica X8

The Electrologica X8, the successor of the Electrologica X1, was advertized to be upwards compatible with the X1, but that was it to some degree only.

Certainly it kept many of the nice features of the X1, and it added new features that made it more fit for scientific calculations and for the implementation of ALGOL 60.

The main store of the X8 was again core store, in units of 16K words of 27 bits. Address space was extended from 2^{15} for the X1 to 2^{18} for the X8. Cycle time was $2.5 \mu\text{sec}$, therefore about 13 times as fast as that of the X1.

The structure of instructions was the same as for the X1: 12 bits operation code and 15 bits address part. Consequently, in a store exceeding 32K not all words could be addressed directly. Again the operation code for most instructions contained $3 * 2$ bits for coding variants.

Registers A and S were again 27 bits long. Now register B was 27 bits long too, thereby frustrating the upwards compatibility X1–X8. Again in the B–variant register B was used as index register, but instead of adding B to the addresspart of the instruction, $B - 16384$ was added, making it possible to access store from address $B - 16384$ to $B + 16383$. This again frustrated X1–X8 compatibility¹. An instruction with the ELAN notation ‘M[B+n]’ as address part was encoded by the assembler to the bit pattern for the instruction with B–variant and with $n + 16384$ as address part.

All instructions for A, S, and B, discussed for the X1, were present again. Multiplication now lasted from 8.75 to $40 \mu\text{sec}$, depending on the number of significant bits of the

¹The X8 console contained a switch by which the X1–X8 compatibility could be augmented. One of its effects was that register B functioned as in the X1.

multiplier, division lasted 40 μsec .

Also all X1 jumps and subroutine calls were maintained, as were the shift instructions. Some shift instructions were much faster, however: the shifts-out for circuits A and S costed at most 5 μsec , whereas the other shifts costed roughly 0.625 μsec per shift place.

The main new features, absent in the X1, were a floating-point register F and instructions for it, some new addressing variants: one to enable the use of registers A, S, F and T (i.e. the instruction counter) as index registers, one for ‘dynamic addressing’ (i.e. two-level addressing), and one for automatic stacking and unstacking of operands and links (using register B as stack pointer), and an ‘execute’ instruction. We give the details below.

4.1 Floating-point instructions

Floating-point register F has 54 bits: a binary exponent of 12 bits, a sign bit and a mantissa of 40 bits². The representation is the Grau representation[7], in which the mantissa is an integer. Denoting the value of the mantissa by m and that of the exponent by e , the value of F is therefore:

$$F = m * 2^e$$

The preferential representation for a value is that with $|e|$ minimal. For F integral with an absolute value of at most $2^{40} - 1$ this implies $e = 0$. While the arguments of the four arithmetic operations (+, -, ×, /) need not to be in preferential form, the result is always standardized to preferential form.

Both exponent and mantissa use one complements representation. For the value of the exponent this implies $-2047 \leq e \leq 2047$, for the mantissa we have $-(2^{40} - 1) \leq m \leq 2^{40} - 1$. For $e = 0$ the preferential representation of the exponent is 12 copies of the sign bit of the mantissa. The absolute value of F is 0 (when $m = 0$) or lies between $0.619 * 10^{-616}$ and $1.777 * 10^{628}$. For negative values of F the exponent is inverted. Hence inversion of F is easy: just replace all zeroes by ones and all ones by zeroes.

The advantage of the Grau representation over more conventional representations (in which the mantissa is a true fraction) is that integer values are represented with binary exponent zero, i.e. in exactly the same way as the normal representation for integers. This makes F extremely fit for handling the mixed-mode expressions of ALGOL 60, without

²The first 27 bits of F contain the binary exponent, the sign bit and the most significant 14 bits of the mantissa, the last 27 bits a bit that is irrelevant for F’s value (mostly a copy of the sign bit) followed by the least significant 26 bits of the mantissa.

any need for change of representation between integers and reals. Also the rounding-off from a real value to an integer one can be carried out very elegantly (see below).

The instructions for register F come in two variants. Denoting the 54 bits of two successive words in store with addresses n and $n + 1$ by $[M[n], M[n+1]]$ we have on one hand the instructions:

ocp	ELAN notation	effect	variants
48	$F + M[n]$	$F := F + [M[n], M[n+1]]$	YN :B PZE
49	$F - M[n]$	$F := F - [M[n], M[n+1]]$	YN :B PZE
50	$F = M[n]$	$F := [M[n], M[n+1]]$	YN :B PZE
51	$F = - M[n]$	$F := - [M[n], M[n+1]]$	YN :B PZE
56	$F \times M[n]$	$F := F \times [M[n], M[n+1]]$	YN :B PZE
57	$F / M[n]$	$F := F / [M[n], M[n+1]]$	YN :B PZE
58	$M[n] = F$	$[M[n], M[n+1]] := F$	YN B PZE
59	$M[n] = - F$	$[M[n], M[n+1]] := - F$	YN B PZE

On the other hand we have the same set of instructions with G instead of F, e.g. $G \times M[n]$. G denotes the ‘tail’ of register F, i.e. the least 26 bits of the mantissa (together with a sign bit). In this case the operand is one word of store. For the first 6 instructions this operand is supplied with 27 copies of its sign bit in order to form a 54 bit operand. In the last two instructions just the tail of F is written to store. All G-instructions have as variants B and PZE only: they cannot be used conditionally.

The arithmetical operations (+, −, ×, /) deliver the best possible result, i.e. the exact value rounded to the nearest representable value. If the exact result lies exactly in the middle of two neighbouring representable values, it is rounded away from zero. There is no underflow or overflow detection.

The rounding of a real value in F, with $-2^{38} - \frac{1}{4} \leq F < 2^{38} + \frac{1}{2}$, to an integral value can be done by the following tric, devised by C.S. Scholten, one of the architects of the X8:

F + Scholten
F − Scholten

where constant Scholten has the value $3 * 2^{38}$. Whether or not this results in an one-word integral value can next be checked by:

U, S = F, Z ” rounding successful?

This instruction tests whether the ‘head’ of F, i.e. the binary exponent and the most significant 14 bits of the mantissa, are all zero (or all one, for negative results).

The execution times of the arithmetical F-operations depend on the value of the two operands. Maximally they last:

$$\begin{aligned} F \pm M[n]: & 18.75 \mu\text{sec}, \\ F \times M[n]: & 68.75 \mu\text{sec}, \text{ and} \\ F / M[n]: & 68.75 \mu\text{sec}. \end{aligned}$$

More details are given in Section 4.6.

Register F can be used for two-word transports in memory, since instructions with opc 50 and 58 do not change the sign bit of the tail to a copy of the sign bit in the head.

4.2 New addressing variants

Before dealing with the new addressing variants we have to discuss some special addresses. In instructions in which an operand is loaded from store the addresses 57 to 62 play a special role: instead of the corresponding store cell one of the registers is used as operand. These ‘addresses’ of the registers are:

57: head of F
 58: tail of F
 59: A
 60: S
 61: B
 62: T

Here T stand for the 18 bits of the instruction counter and the contents of a number of the one-bit registers. Especially the sign bit is filled with condition register C (this on special demand of the ALGOL 60 implementation team).

In ELAN just the names of the registers can be used. Therefore we could write $U, S = F, Z$ above, which is encoded by the assembler as $U, S = M[57], Z$. The instructions of the X1 for copying or inverting register values are, in view of the X1–X8 compatibility, probably present in the X8, but are never used and do not have an ELAN notation.

Another special address is 63. $M[63]$ is called ‘D’, is on the one hand a normal store location, but plays, as we will see in a moment, a special role in most of the new addressing variants.

The new addressing variant is called ‘DYN’, for dynamic. In this variant the 15 bits of the address part of an instruction are split in two groups, one of 6 bits and the other of 9 bits. Let p denote the value of the group of 6 bits ($0 \leq p \leq 63$) and q the value of the group of 9 bits ($-256 \leq q \leq 255$)³.

For $0 \leq p \leq 57$ the ELAN notation $Mp[q]$ is used. $Mp[q]$ addresses the store cell with address $M[D'+p]' + q$, where $D = M[63]$ and the primes indicate that in the address calculation only the least significant 18 bits of a word are taken into account (the remaining bits being replaced by copies of the sign bit). This address variant enables a direct implementation of two-level addresses as required by the block structure of ALGOL 60. It extends execution time of an instruction by 5 μ sec, the time for two additional store cycles: one for accessing D and one for accessing $M[D'+p]$.

For $p = 63$ the ELAN notation $MD[q]$ is used. $MD[q]$ addresses the store cell with address $D' + q$. It costs one additional store cycle.

The ELAN notations for $p = 58, 59, 60, 61,$ and 62 are $MG[q], MA[q], MS[q], MC[q],$ and $MT[q]$. Here $MG[q], MA[q], MS[q],$ and $MT[q]$ address the store cells with addresses $G' + q, A' + q, S' + q,$ and $T' + q,$ respectively. $MC[q]$ addresses the store cell with address $B' + q,$ but with a side effect:

- if an operand is loaded from store, B is decreased by 1 or 2 afterwards,
- if an operand is written to store, B is incremented by 1 or 2 afterwards.

The change of B is 1 for $A-, S-, B-,$ and $G-$ instructions (having a one-word operand) and 2 for $F-$ instructions (having a two-word operand). This facility makes it possible to use B as a stack pointer, with automatic adaptation for stack instructions.

If $q = 0,$ ELAN allows to omit the $[q]$ part from the instruction notation. Hence, to stack the value of F one simply writes $MC = F.$

The DYN-variant makes it possible to address also store locations with addresses from 2^{15} to $2^{18} - 1.$

This DYN-variant is available for most X8-instructions.

Besides this DYN-variant there exists a :DYN-variant, in which not the operand is loaded, but only its address (in case of the :MC-variant without side-effect). This variant is available for less instructions. There is a subtle difference between instruction $S = A$ and instruction $S = :MA.$ In the first case all bits of A are copied to $S,$ in the second case its least significant 18 bits plus 9 copies of the sign bit.

³In the instruction q is encoded by $q + 256.$

4.3 A new subroutine call

Besides the 16 subroutine calls that were already present in the X1 and had fixed addresses for their links the X8 has one new subroutine call:

ELAN notation	variants
SUBC(label)	UYN : B DYN :DYN

It puts its link in the store location with address B' and increments B by 1, in other words, it puts its link on top of stack. This makes the instruction fit for implementation of recursive routines. In contrast to the old subroutine calls of the X1, the address part of the instruction is, except for the :- and the :DYN-variants, not the subroutine's begin address, but the address of the store location in which that begin address can be found.

In addition there exists a variant of SUBC, called SUBCD, which makes the X8 deaf for interrupt signals. It is used inside operating systems only.

4.4 The execute instructions

In order to simplify parameter handling, an execute instruction was added to the X8. Its ELAN notation is:

DO(label)

with the effect that the instruction in M[label] is executed, whereafter program execution is continued by the instruction following the DO. If M[label] happens to contain a subroutine call, the link points to the instruction following DO.

During the development of the ALGOL 60 implementation the team concluded that occasionally also access to the address of the instruction executed by DO was necessary. Therefore they proposed to Electrologica to add a variant of DO, called DOS, which as a side effect loads that address in register S.

Both DO and DOS allow the variants UYN : B DYN :DYN.

4.5 Some other additions to the X8 instruction set

In the first place we have the instructions:

ELAN notation	effect	variants
PLUSA(label)	store[label]:= A:= store[label] + A	PZE
MINA(label)	store[label]:= A:= store[label] - A	PZE

with analogous instructions PLUSB, MINB, PLUSC, and MINC for register S and B.

Moreover the structure of input/output differs completely from that of the X1. The X8 has a separate input/output processor, capable of executing complex transport tasks autonomously. There are instructions for regulating the interaction between X8 and the input/output processor. Since such instructions are part of an operating system and do not occur in the ALGOL 60 implementation itself, they are of no further interest here.

4.6 The execution times of floating-point operations

In order to be able to carry out numerous kinds of measurements of the ALGOL 60-implementation for the EL X8 I wrote an X8-emulator in Pascal. I also included the execution times for all instructions as described in the official programmers guide of Electrologica.

For most instructions this description was rather complete. We give some examples⁴:

instruction	time(μ sec)	condition
A + n	2.50	
A + M[n]	5.00	
A + M[B+n]	5.00	
A + :MA[q]	3.75	
A + MA[q]	5.00	
A + MD[q]	7.50	
A + :MD[q]	6.25	
A + Mp[q]	10.00	
A + :Mp[q]	8.75	
LUA(n)	3.75	n < 16
LUA(n)	5.00	n \geq 16
LCA(0)	3.75	
LCA(n)	$\lfloor (n+5)/2 \rfloor$	n > 0

⁴LUA(n) shifts A out to the left over n places, LCA(n) shifts A circular to the left over n places.

For the instructions MULAS, MULS, and the floating-point operations, however, ranges are given for the execution times. For MULAS we have:

$$\text{MULAS}(n): (5.00 + y) \mu\text{sec}, \text{ with } 1.25 \leq y \leq 32.5$$

and the same range for MULS. Digging deeply in memory I faintly remembered that multiplication time depends on the number of significant bits of the multiplier. Therefore we adopted the following model: each multiplication step costs $1.25 \mu\text{sec}$, and the number of steps is at least 1 and at most 26, depending on the number of significant bits of S.

For floating-point addition and subtraction the following figures are given by Electrologica:

instruction	time(μsec)	condition
F + n	3.75	exponent of F zero
G + M[n]	5.00	exponent of F zero
F + M[n]	7.50	both exponents zero
F + n	6.25 – 15.00	otherwise
G + M[n]	7.50 – 17.50	otherwise
F + M[n]	8.75 – 18.75	otherwise

In general, there are four phases for the execution of an addition: reading and decoding the instruction, obtaining the (second) operand, comparing and equalizing the binary exponents by shifting zero, one, or both operands, and finally bringing the result in normal form. In view of the fact that the execution time is variable only when not both binary exponents are zero I attributed the variation mainly to the process of making the two exponents equal. Excluding some special cases, like a zero mantissa or the exponents differing more than 81 in absolute value, we first try to shift the mantissa corresponding to the largest exponent to the left (thus reducing that exponent) and thereafter we shift, if still necessary, the other mantissa to the right, thereby losing information. Left shifts are carried out over at most 39 positions, shifts to the right over at most 41 positions⁵. We assumed that each shift takes between 0 and 3 cycles of $1\frac{1}{4}\mu\text{sec}$.

For floating-point multiplication and division the following figures are given by Electrologica:

⁵For the purpose of correct rounding-off of the result of an floating-point operation register F had two guarding bits.

instruction	time(μ sec)
$F \times n$	10.00 – 66.25
$G \times M[n]$	11.25 – 67.50
$F \times M[n]$	12.50 – 68.75
F / n	60.00 – 66.25
$G / M[n]$	61.25 – 67.50
$F / M[n]$	62.50 – 68.75

For multiplication we assumed that the variation in execution time depends on the number of multiplication steps (of 1.25 μ sec each), i.e. on the number of significant bits of the multiplier⁶. Since one of the operands in $F \times n$ had at most 15 significant bits and in $G \times M[n]$ at most 26, we expect for $F \times n$ rather 10.00 – 35.00 and for $F \times G[n]$ rather 11.25 – 50.00 as ranges.

For division part of the variation is probably caused by shifting the mantissa of divider or dividend to the left such their quotient is at least 1 and smaller than 2.

Whenever we specify X8 execution times for program parts in the next chapters and in the appendices, it is with the proviso that the timing model used in our X8 emulator is correct. Moreover we did not take into account any slowing down of execution times by the cycle stealing done by CHARON, the separate input/output processor of the X8.

⁶If I do remember it correctly, multiplier and multiplicand were interchanged if the latter had a smaller number of significant bits

Chapter 5

The ALGOL 60 system for the EL X8

5.1 Introduction

The project to develop an ALGOL 60 implementation for the EL 8 started 1963. The team, initially consisting of Zonneveld, Van de Laarschot, Barning, Nederkoorn, and myself, was rather ambitious.

In the first place we aimed at implementing ALGOL 60 without any restriction. This included arbitrary order of declarations, allowance to omit specifications of formal parameters called by name, integer labels, and, of course, recursion. There were only a few places where we deviated from the semantics of the Revised Report. So we interpreted access to a switch designator with an index out of bounds as a run-time error.

Of course we planned the implementation of dynamic own arrays. These could have variable size: at re-entrance of the block in which an own array is declared its bounds are re-evaluated yielding possibly values that differ from their previous ones. Hence they had to be allocated on a heap (we used the term ‘contra-stack’).

We also planned the extension of the language with a string type as first-class citizen (simple and subscripted string variables, string expressions, string assignments, and functions delivering a string result). Here again the variable size of strings demanded the use of a heap.

The compiler should carry out a thorough check on the syntactical correctness of source programs. At run time as much semantical checks should be carried out as was feasible

without slowing down execution speed exceedingly. This included some dynamic tests on the correspondence between actual and formal parameters.

Nederkoorn and I defined the mapping of source programs to object programs including these extensions. The subroutines to be called in an object program were also written, with the exclusion of those handling the declaration of own arrays and all other heap operations. Those we postponed temporarily, and in the end they were never written. The compiler, however, generates the mapping as defined by Nederkoorn and myself and it is at run time only that own arrays and string operations lead to a run-time error message ('... not implemented').

Some language constructions, such as integer labels, own arrays, and our string-type extensions, lead to a certain amount of execution overhead. We were, however, careful *not* to burden programs not using such constructions with this overhead.

Another example hereof is the implementation of labels. In general a two-word variable, called label variable, was reserved and initialized for each label when entering the block to which that label was local. For labels that were used in local goto statements only (or were not used at all) no label variables were introduced.

After the definition of the mapping from source language to object code and the design of the run-time support routines we compiled a number of small test programs by hand and tested them by means of the X8-emulator written by Barning. No amendments showed necessary. Only after that work I started to develop the compiler.

Although the mapping from source language to object code was defined prior to and independently of the compiler design, its structure, with a restricted amount of optimization, was simple enough to allow a direct implementation of the compiler. It was at only one place (in the step-until-element) convenient to change the order of the object-code instructions as defined in the mapping in order to simplify (the translator scan of) the compiler.

To us it was self-evident that the system would be a load-and-go system never storing object programs externally. We had in mind to transform the X8 into a machine running ALGOL 60 programs sequentially without much interference of an operator. In the user manual[10] that I wrote in 1965 on request of Electrologica, the fact that source programs were compiled before execution was not mentioned at all, as being irrelevant to the user. It confined itself to stating that programs were checked against syntax errors and only executed if faultless.

As we will see in the next sections, there are a number of differences between the ALGOL 60-system of the EL X1 and that of the EL X8. They concern the use of registers as

‘top of stack’, the representation of values, the use of X8-instructions in object programs instead of pseudo P-code as was the case for the X1, the amount of validity checks at both compile time and run time, and the structure of the compiler.

In contrast with the ALGOL 60-system of the X1, all results of expressions and all addresses are delivered in a register and *not* on top of stack. Such a result or address is written to the stack if and only if the register involved is temporarily necessary for another value.

One of the major design decisions was to give some of the registers each a special role:

- register B is used almost exclusively as stack pointer;
- all arithmetic values, both of real and of integer type, are built in register F;
- Boolean results are constructed in register C (the condition register);
- all addresses, both of arrays and of label variables, are delivered in register A;
- the major role of register S is as working register.

This arrangement was rather useful, also in exceptional cases. An example of the latter was the object code for integers as actual parameter of a procedure for which an interpretation both as arithmetic value and as label value was possible. Evaluation of the corresponding formal parameter then delivered its numeric value in register F and its label value in register A. More details are given in Section 5.2.6.

The value representation of variables was rather direct: integers in one word, reals in two words, simple booleans in one word (**true** represented by 0 and **false** by 1), boolean array elements in one bit (packed 27 elements per word), and strings in one word (represented by a heap address).

There was, indeed, a ‘complex’ of run-time support routines, but it had a role different from that of the X1. Wherever sensible the X8 object program was coded directly in terms of X8 instructions, and only fixed patterns consisting of several instructions were coded as subroutine in the complex for the sake of reducing object-program size. Here we see a compromise between execution speed and program length.

The structure of the compiler was quite different, containing, in modern terms, a recursive descent parser. This made it rather easy to incorporate a thorough syntax check and also to include an effective ‘back-on-the-rails’ procedure.

5.2 The structure of the object program

5.2.1 Simple assignment statements

As was the case with ALGOL implementation for the X1, we start our discussion for the X8 case with the example statement:

$$i := i + 1$$

In case i is a variable of type integer declared outside procedures it has again a fixed location in store. Let its (static) address be 4880. Then the object code reads:

instruction	explanation
G = M[4880]	Take Integer Value of i
F + 1	Add Small Integer Constant 1
S = F, Z]
N, SUBC(:RND)] Store Integer in i
M[4880] = G]

During program execution this requires the execution of 5 instructions and takes 20 μsec ¹.

From this example we see that arithmetic expressions are evaluated in register F. The stack is only used when an intermediate result in F should be saved, in order that F can be used for the evaluation of another subexpression. Moreover, the address and type of the left hand side of this assignment statement is kept in the compiler for use after the generation of the code for the right hand side.

In case variable i is local to a procedure body, the translation of the statement ' $i := i + 1$ ' reads:

instruction	explanation
G = MD[5]	Take Integer Value of i
F + 1	Add Small Integer Constant 1
S = F, Z]
N, SUBC(:RND)] Store Integer in i
MD[5] = G]

¹provided that no integer overflow occurs and the third instruction sets condition register C to true.

due to the dynamic addressing² of i . Now its execution takes 25 μsec .

The code for statement

$$x := x + y$$

with x and y variables of type real, declared in the outermost block simply reads:

instruction	explanation
F = M[@x]	Take Real Value of x
F + M[@y]	Add Real Value of y
M[@x] = F	Store Real in x

Execution time depends on the values of x and y and is minimally 22.50 μsec and maximally 33.75 μsec . For initial values 3.14 of x and 0.1 of y the execution time is 26.25 μsec ³.

The statements $x := x \times y$ and $x := x/y$ are coded analogously and (for the values of x and y as above) executed in 77.50 and 78.75 μsec , respectively.

5.2.2 Array access

We now turn to array access. As was the case for the ALGOL 60 implementation for the X1, for each array a storage function is constructed at declaration time. Now its length is $3 * n + 4$, where n is the array dimension, such that it can contain the lower and upper bound for each index position. Moreover it contains the array type and the address of the area in store reserved for the elements. The address of this storage function is stored in a pseudo-variable local to the block in which the array declaration occurs, called the array key. This kind of indirection is useful in compiling a formal array identifier called by value, where the compiler has no knowledge of the dimension of the actual parameter (this problem was solved in X1-ALGOL by restricting that dimension to at most 5).

Let R be a one-dimensional real array, x a real variable and i a variable of integer type, all declared outside procedures. Then the object code for:

²If i is declared local to a procedure, it gets a two-component address (block number, displacement). If block number = 2 and displacement = 5, one would expect M2[5] as dynamic address. Since MD[2] = D, this is optimized to MD[5].

³All timing figures for floating-point operations are modulo the correctness of our timing model for the floating-point instructions of the X8.

$$R[i] := R[i] + x$$

reads:

instruction	explanation
G = M[4880]	Take Integer Value of i
A = M[4883]	Take Array Key of R
MC = F	Stack F
MC = A	Stack A
G = M[4880]	Take Integer Value of i
A = M[4883]	Take Array Key of R
SUB0(:IND)]
F = MA] Take Subscripted Real
F + M[4881]	Add Real Value of x
SUB2(:STSR)	Store Subscripted Real

Now the translation of the left-hand side of the assignment consists of $4 + 1$ instructions. The first 4 instructions put the index value and the address of the storage function of the array on top of stack. Thereafter F and A are available for the evaluation of the right-hand side of the assignment, for which 5 instructions are generated. The last instruction, a subroutine call to the complex, finds in F the value of the expression and on top of stack the address description for the left-hand side. It computes the address of the array element and carries out the assignment. It uses the same complex routine 'IND' to convert an index in F and an array key in A into the address of the array element in A. Within IND the index (or indices, for an array of higher dimension) is (are) checked against the lower and upper bound(s) of the array.

In the ALGOL 60 implementation for the X1 the address of a left-hand side array element is elaborated before evaluating the right-hand side expression. In X8 ALGOL this is postponed to the assignment operation itself. This was done for two reasons:

1. If the left-hand side variable is an element of an *own* array, it could happen that during the evaluation of the right-hand side expression both location and bounds of the array change. This possibility arises if the own array is declared with dynamic bounds locally in a function and the expression contains a (recursive) call of that function, changing thereby the array bounds and giving the array possibly a new location in the heap⁴. By

⁴Here we used the interpretation of ALGOL 60 in which all incarnations of a recursive procedure or function share one and the same copy of a locally declared own array.

applying IND after the evaluation of the expression the indices can be checked against the (possibly new) bounds and the (possibly new) address of the element can be calculated.

2. A second reason is that thereby the storage function is still available, containing type information. This is used in the case that the array identifier of the left-hand side variable is a formal identifier of (a to the compiler) unknown type.

Execution of the above piece of object code of 10 instructions gives rise to the execution of 41 instructions and costs from 197.50 to maximally 208.75 μsec . For $R[i] = 0.1$ and $x = 3.14$ the cost is 201.25 μsec .

For the implementation of ALGOL 60 for the X8 there is a difference in execution time for the statements 'I[i]:= 0' and B[i]:= **true**, where I is an integer array and B a boolean array. This is caused by the fact that boolean-array elements are represented by bits, 27 elements per X8 word, and therefore one bit in a word has to be cleared and set. We give the code for the subroutine ST(ore) S(ubscripted) B(olean):

STSB:	A = MC[-1]	" A:= address of storage function
	F = MC[-2]	" F:= last index
	SUB1(:INDB)	" INDB does not disturb C!
		" address word in A, a 1 at bit position in S
	stock3 = S	" save S, having a 1 at the right bit position
	S = - S	" now S has a 0 at the right position
	S '×' MA	" clear old value
	N, S '+' stock3	" substitute new value
	MA = S	" store word with new value substituted
	GOTO(M[10])	" return

with IND(exer for) B(olean arrays) is given by⁵:

⁵INDB delivers in register F a value 2 as type indication Boolean. That is used in some subroutines but not in STSB.

INDB:	SUB0(:IND)	" index routine
	S = A	" delivers (27 * word address + bit position) in A
	A = 0	" clear A
	DIVAS(27)	" S:= word address, A:= bit position
	F = :MC	" save stack pointer in F
	B = :MA	" B:= bit position in word
	A = :MS	" A:= address of word containing the bit
	S = 1	" S:= one bit 1 at position 0
	LCS(B)	" shift S circular to the left over B positions
	B = :MG	" restore stack pointer from F
	F = 2	" type indication Boolean
	GOTOR(M[9])	" return, restoring condition C from the link

Consequently, the assignment $I[i]:= 0$ costs $120 \mu\text{sec}$ whereas the assignment $B[i]:= \mathbf{true}$ takes $192.5 \mu\text{sec}$.

5.2.3 The <for-statement>

The object code for a for-statement in the X8-version is rather direct, certainly compared with its X1-counterpart. For the statement

for i:= 1 **step** 1 **until** n **do** S

it reads:

label	instruction	comment
	F = :L2	"] store address of step
	M[...] = G	"] in for-list variable
	F = 1	" initial value for <i>i</i>
	GOTO(:L3)	" store in <i>i</i> and test
L1:	F = 1	" step size
L2:	G = M[@i]	" Take Integer Value of <i>i</i>
	MC = F	" Stack (old) value of <i>i</i>
	DO(L1)	" load step size
	F + MC[-2]	" add value of <i>i</i>
L3:	S = F, Z	"]
	N, SUBC(:RND)	"] Store Integer in <i>i</i>
	M[@i] = G	"]
	MC = F	" Stack (new) value of <i>i</i>
	G = M[@n]	" Take Integer Value of <i>n</i>
	F - MC[-2], Z	" equal to the value of <i>i</i> ?
	DO(L1)	" load step size
	N, F = F, E	" otherwise, do step size and <i>i</i> - <i>n</i> have the same sign?
	N, F = F, Z	" or step size = 0?
	Y, GOTO(:L4)	" then execute statement S
	GOTO(:L5)	" jump to code following for-statement
L4:	...	" object code for S
	GOTO(M[...])	" jump via for-list variable to continue iteration
L5:	...	"

In the case that the controlled variable is a simple non-formal variable we can distinguish the following three parts:

1. the translation for the for-list-elements (here only one), each beginning with two instructions by setting a variable (called for-list variable) to the label (here L2) where the execution has to be continued after execution of statement S. This part is concluded by a jump over Part 2 (here to label L5).

The code for a step-until-element itself has several subparts:

- a. code for loading the initial value into F, followed by a jump over subparts b and c to subpart d,
- b. code for loading the step size into F. If it consists of more than one instruction, it is followed by a return instruction (i.e. GOTO(MC[-1])),
- c. code for adding the value of the controlled variable and the step size. A step size

compiled to one instruction is loaded by a DO-instruction, executing that single instruction, otherwise part b is called as a subroutine,

d. code for assigning the value of F to the controlled variable and testing that value against the upper bound. It is concluded by a conditional jump to Part 2, i.e. the code for statement S.

2. code for statement S.

3. a jump, via the for-list variable, to the relevant for-list-element.

There is no block introduction. In each block the compiler reserves a number of words for accomodating for-list variables, as many, in fact, as necessary for giving each for-statement in the deepest nesting of for-statements in the block its own variable (for-list variables are shared between non-nested for-statements).

Execution of a for-statement begins by executing the first for-list-element. In our example, it begins by assigning label L2 to the for-list variable, loading initial value 1 to F and jumping to Label L3 (i.e. Subpart 1.d), in order to assign it to i and to test whether or not the step-until element is exhausted. If not, control is moved to Label L4 (i.e. Part 2). After execution of statement S a jump is carried out via the for-list variable to Label L2, where the value of i is incremented by step size 1, before we again arrive at Label L3. After exhaustion of the step-until element the code for the next element is entered. After completion of the last element a jump over Part 2 (here to Label L5) is executed, completing the execution of the for-statement.

Taking for S the <dummy statement>, with an object code of 0 instructions, execution of statement

for $i := 1$ step 1 until n do S

takes:

for $n > 0$: $66.25 + n * 83.75 \mu\text{sec}$,

for $n \leq 0$: $71.25 \mu\text{sec}$.

In case that the controlled variable is an array element (occurring seldom, but not excluded in ALGOL 60) or a formal identifier (occurring regularly in the so-called Jensen's device) the object code is slightly more complicated to contain code for the preparation of assignments to the controlled variable. Still it remains extremely simple and direct compared to the code for the X1. In after-thought, however, some optimizations for the simplest cases can easily be conceived.

5.2.4 Procedure calls

Let again procedure p be declared in the outermost block by:

```
procedure p(z); real z;
begin real y;
end;
```

and let x be a simple global variable of type real with initial value 3.14.

Then the call $p(x)$ generates the following object code:

SUBC(@p)	" subroutine call to the object code of p 's declaration
(0 + @x)	" parameter descriptor for x

with resulting execution time $182.5 \mu\text{sec}$, whereas the call $p(x+0.1)$ produces:

GOTO(:L1)	" jump over implicit subroutine
L0: SUB2(:ENTRIS)	" enter implicit subroutine
F = M[@x]	" F:= value of x
F + M[@0.1]	" F:= F + 0.1
GOTO(:EXITIS)	" exit implicit subroutine
L1: SUBC(@p)	" subroutine call to the object code of p 's declaration
(20 * 2 ²⁰ + @L0)	" parameter descriptor for $x + 0.1$

with execution time $185 \mu\text{sec}$ (due to the jump over the implicit subroutine, adding $2.5 \mu\text{sec}$).

In the context of procedure p_1 , declared in the outermost block by:

```
procedure p1(z); real z;
begin real y;
  y:= z
end;
```

the call $p_1(x)$ executes in $207.5 \mu\text{sec}$ (i.e. $25 \mu\text{sec}$ for statement $y:= z$) and $p_1(x+0.1)$ in $278.75 \mu\text{sec}$ (i.e. $93.75 \mu\text{sec}$ for $y:= z$).

For an explication of these figures we need again to go into some details of the addressing mechanism of the ALGOL 60 implementation for the X8.

Instead of one display at a fixed location in store which has to be updated at the occasion of each context switch, now each procedure incarnation has its own display at a fixed

location in its block cell in the stack. The begin address of this display is stored in $M[63]$, hence the name D for that location. As we have seen in the previous chapter, $M[63]$ plays a central role in the dynamic-addressing variant of instructions: using that variant the transformation of a two-level address to a (static) store location is carried out by hardware.

The cost of the construction of a display is paid at procedure entrance. The benefits show up by all other context switches, such as entrance and exit of an implicit subroutine and procedure exit, either by completion or by a goto-statement. In these cases simply the begin address of an old display has to be assigned to D .

The code for the declaration of $p1$ reads:

p1: S = D	" S:= current value of D
A = - 0, Z	" A:= block number - 2
SUB0(:DPTR)	" Display Transport
B + 1	" reserve one word for new block cell pointer
SUB0(:CEN)	" Call Expression by Name
F = :MA[2]	" F:= address of word reserved in display
B + 2	" reserve two words for local variable y
SUB2(:ENTRPB)	" Enter Procedure Body
DOS(MD[3])	" execute first parameter word: F:= z
MD[5]:= F	" store result in y
GOTO(:EXITP)	" Exit Procedure

In this figure we can discern 5 sections. The first one, of 4 instructions, constructs the new display. The next section contains a subroutine call for each formal parameter. For parameters in the value list specified **real**, **integer**, **boolean**, or **label** subroutines CRV (Call Real by Value), CIV (Call Integer by Value), CBV (Call Boolean by Value), or CLV (Call Label by Value) are called. For name parameters occurring in the procedure body as left part of an assignment statement subroutine CLPN (Call Left Part by Name) is called⁶ and for all other parameters subroutine CEN (Call Expression by Name). CRV, CIV, CBV, and CLV just put the value of the actual parameter on top of the stack, CEN constructs a two-word parameter descriptor and CLPN a four-word descriptor on top of the stack.

The two-word descriptor constructed by CEN consists of an instruction which, when executed, evaluates the actual parameter, and in the second word the display pointer of

⁶CLPN checks dynamically whether the corresponding actual parameter is a variable.

the calling context and a copy of that part of the parameter descriptor in the code that encodes the nature of the actual parameter. Instead of 4 cases, as we saw for the X1 system, now even 32 different possibilities are distinguished.

The four-word descriptor constructed by CLPN contains two additional instructions for use in assignments to the formal parameter: one to be executed before the evaluation of the right hand side of the assignment statement (i.e. the expression), the other one after that evaluation to accomplish the assignment itself. If the corresponding actual parameter is a simple variable, the former instruction is an innocent one and the latter either a simple store instruction (for real variables) or the call of a subroutine of the complex (in the integer or boolean case). On the other hand, if the actual parameter is a subscripted variable, the former instruction is a special call of the implicit subroutine constructing an address description on the stack, whereas the latter instruction is always the call of a subroutine of the complex.

In Section 3.3.4. we saw that in the X1 system the construction of a parameter descriptor on the stack is carried out by ETMP *before* control is transferred to the code of the procedure declaration. In the X8 system that is done after that transfer of control, by the code for the procedure body itself. The reason for this is that at the calling environment the information whether a two-word or a four-word descriptor has to be constructed is not necessarily available⁷.

The third section of the code for the declaration of p1, 3 instructions long, completes the entrance of the procedure. Next comes the section with the code for the statements of the procedure body, here the statement $y:=z$. The DOS instruction executes the instruction in the parameter descriptor constructed by CEN. In the call p1(x) it is the instruction 'F = M[@x]', in the call p1(x+0.1) it is the instruction 'SUBC(:L0)', where L0 is the begin address of the implicit subroutine preceding the procedure call. As a side effect of the DOS instruction the address of the two-word parameter descriptor is delivered in register S. The implicit subroutine starts with a call of ENTRIS, in which the current value of D (the context of the procedure body) is saved in the stack and replaced by the value of the second word of the two-word parameter descriptor, i.e. the context of the procedure call. The implicit subroutine ends with a jump to EXITIS, in which the context is set back to the procedure body (by means of the value of D saved by ENTRIS) whereafter control is transferred to the instruction following DOS.

The last section of the code for the procedure declaration is the procedure exit.

⁷Since the construction of a four-word descriptor costs additional execution time, it is, as a kind of optimization, avoided when not necessary.

For procedure p2, declared in the outermost block by:

```

procedure p2(z); real z;
begin real y;
    z:= z + 0.1
end;

```

the following code is generated:

p2: S = D	" S:= current value of D
A = - 0, Z	" A:= block number - 2
SUB0(:DPTR)	" Display Transport
B + 1	" reserve one word for new block cell pointer
SUB1(:CLPN)	" Call Left Part by Name
F = :MA[2]	" F:= address of word reserved in display
B + 2	" reserve two words for local variable y
SUB2(:ENTRPB)	" Enter Procedure Body
DOS(MD[5])	" prepare assignment to z
DOS(MD[3])	" F:= z
F + M[@0.1]	" F:= F + 0.1
DOS(MD[6])	" assign value of F to z
GOTO(:EXITP)	" Exit Procedure

Execution of a call p2(x) now takes 307,5 μ sec, whereof 148.75 μ sec for the call of CLPN and 51.25 μ sec for the statement z:= z + 0.1 (the call of CEN occurring in the execution of p(x) costs 75 μ sec).

5.2.5 Some standard functions

The standard functions *sqrt*, *sin*, *cos*, *ln*, and *exp* were, as was the case for the X1, implemented as subroutines of the complex, transforming the top-of-stack value. On the X8 also function *arctan* was implemented that way. The object code for statement

$$x := \sin(y)$$

reads simply:

F = M[@y]	” F:= value of y
SUBC(:SIN)	” call of the sine routine, F:= sin(F)
M[@x] = F	” x:= F

In contrast to the X1 implementation all standard-function identifiers can be used as actual parameters. The call

$$p3(\sin)$$

of procedure $p3$ with heading

procedure p3(f); real procedure f;

reads:

JU(:L1)	” jump over implicit subroutine to L1
L0: S = MS[1]	” Take Formal Display pointer
GOTO(COS L)	” jump indirectly to routine SIN P
L1: SUBC(@p3)	” subroutine call to the object code of p3
(24 * 2 ²⁰ + :L0)	” parameter descriptor for implicit subroutine

SIN P is the subroutine of the complex to which control is passed whenever $p3$'s parameter f is used as function identifier in a function designator. It first evaluates the actual parameter of the function designator (delivering the result in F) and then jumps to the sine routine. Where the computation of $\sin(0.3)$ takes $635.00 \mu\text{sec}$, the computation of $f(0.3)$ costs $777.50 \mu\text{sec}$. In this way we combined efficiency (for normal use of a standard function) with generality (allowing its use as an actual parameter).

Again we give some data on the execution performance. They were measured as the average for a range of values. For sqrt , exp , ln , and arctan we used the arguments 1.0 (1.0) 20.0, for sin and cos the values $-\pi$ ($\pi/10$) $+\pi$.

function	instructions	time (μsec)
<i>sqrt</i>	45	343.75
<i>sin</i>	32	656.25
<i>cos</i>	33	603.75
<i>exp</i>	76	985.00
<i>ln</i>	58	700.00
<i>arctan</i>	61	785.19

5.2.6 Designational expressions and goto statements

As was already mentioned in Section 5.1, a two-word label variable corresponded to each label. It was reserved in the local space of the block to which that label was local, each time the block was entered. It was initialized with the object-program address of the label, the current display pointer and the blocknumber of the block. The object code for a statement:

goto if b then aa else bb

read:

S = M[@b], Z	" load value of b in condition register C
N, GOTO(:L0)	" conditional jump to else-part
A = @aa	" load address of variable for aa in register A
GOTO(:L1)	" jump over else-part
L0: A = @bb	" load address of variable for bb in register A
L1: GOTO(:JUA)	" JUA

where the code of routine JUA in the complex of subroutines accesses the label variable to produce a context switch (by installing a new display pointer and adapting the stack pointer) and to execute the jump to the object-program address.

Statement:

goto aa

for local label aa is, however, translated simply by:

GOTO(:Laa) " jump to aa

where Laa is the object-program address of aa⁸.

The evaluation of a designational expression that is used as actual parameter to a procedure always yields the address of a label variable. If the corresponding formal parameter is a value parameter, the contents of that label variable are copied into the parameter area of the procedure by the subroutine CLV (Call Label by Value, c.f. Section 5.2.4.). The introduction of label variables made the implementation of value calls of labels an easy job.

⁸If label aa is used only in such simple, local goto statements, the reservation and initialization of a label variable for aa is suppressed, saving execution time.

Statement:

$p(aa)$

with aa some (nonformal) label and p some (nonformal) procedure, is translated as:

$SUBC(@p)$ " subroutine call to the object code for p $(14 * 2^{20} + @aa)$ " parameter descriptor for aa 's label variable
--

The translation of procedure statement $p(2)$ in a context where 2 can also be interpreted as an integer label reads⁹:

$GOTO(:L1)$ " jump over implicit subroutine L0: $SUB2(:ENTRIS)$ " enter implicit subroutine $F = 2$ " load value 2 in register F $A = @2$ " load address of label variable for 2 in A $GOTO(:EXITIS)$ " exit implicit subroutine L1: $SUBC(@p)$ " subroutine call to the object code for p $(29 * 2^{20} + @L0)$ " parameter descriptor for ambivalent 2
--

Here we profit from the fact that different value types use different registers. This implementation of ambivalent integer parameters allows that procedure q with declaration:

procedure $q(p,x); p(x);$

can be called both by statement $q(p1,2)$ as by statement $q(p2,2)$, where $p1$ and $p2$ are declared by:

procedeu $p1(x); goto x;$
procedure $p2(x); i:= i + x;$

and 2 is in scope as integer label for both $p1$ and $p2$ ¹⁰.

⁹The compiler translates actual parameters of a procedure statement independently of the procedure's declaration. This makes sense since in the case of the use of formal procedures that declaration might be unknown or not unique.

¹⁰A declaration '**procedure** $p(x); if b then goto x else i:= i + x;$ ' leads to an error message of the compiler, complaining (probably in conflict with the Revised Report) about inconsistent use of parameter x .

5.2.7 Line numbers

After the design of the object code by Nederkoorn and Kruseman Aretz and, if my remembrance is correct, even after the design of the compiler in ALGOL 60, the idea was born that it would be helpful to users of the system if in case of a run-time error they were informed about the place in the source program where the error was detected. This, however, was not easily implemented in the existing design. Moreover, I did not like to add a long table to the object program with a mapping from instruction counters to source code positions. Therefore I devised a cheaper solution, which gave the users the line number of the most recently started statement.

For that purpose the object program was larded with instruction pairs of the form:

```
S = ...
line counter = S  " set line counter
```

execution of which takes 7.5 μ sec.

In principle these instructions are generated as the begin of the translation of an ALGOL 60 statement, unless it is certain to the compiler that the value of *line counter* has already the correct value. For the source program line:

```
begin i:= 0; j:= 0; A[i,j]:= 1 end;
```

the instruction pair is generated only once. The same is true for the following lines:

```
if i = n then ready:= true;  
for i:= 1 step 1 until n do B[i]:= 0;
```

There are complications, however. For the statement:

```
for i:= 1 step 1 until n do  
  begin x:= x + A[i];  
    A[i]:= 0  
  end;
```

there are generated 4 instruction pairs, the last one just prior to the indirect backwards jump to the for-list element.

Another complication is the use of a function designator within an expression. After the elaboration of the actual parameters (in which the position of the call is relevant) the current value of *line counter* is saved to the stack (in the newly constructed block cell) before the function body is executed. At the end of the function call *line counter* is reset from the stack to the value relevant for the calling environment.

Also in front of array declarations the line-number setting instruction pair is generated.

I made an effort to restrict the overhead of line-number setting during program execution as much as possible. The code for generating the necessary instructions is scattered throughout the compiler and the complete algorithm is rather complex. A slightly improved version is treated in [13]. It was made possible to request the system to produce an object program without line-number administration during execution, leading to some savings in execution time and object-program length.

In 1965 I estimated that those saving would be in the order of a few percents. I never measured it. Only recently I did measurements for a number of programs. Those programs are dealt with in the next chapter. The results read:

program	with line numbers		without line numbers		code ratio	time ratio
	code	time	code	time		
Havie integrator	609	96.70	507	96.20	1.20	1.005
QR eigenvalues	1211	0.2850	1047	0.2785	1.16	1.023
JAZZ164	2497	10.36	2165	10.22	1.15	1.014
Erathostenes	208	6.347	176	6.003	1.18	1.057
pentomino	1019	51.43	839	48.36	1.21	1.063
lisp interpreter	5128	636.7	4052	597.8	1.27	1.065

In program length the savings vary from 15 to 27%, in execution time from 0.5 to 6.5%, more than I expected. When we discuss the programs in more detail in the next chapter, we will also go into the question why the savings vary so much as they do.

5.2.8 Guarding stack overflow

Where in the X1 implementation of ALGOL 60 the execution stack could grow unlimited and could overwrite object program and run-time support routines without any warning, in the X8 implementation stack growth is guarded against overflow. The implementation is such that the test on stack overflow is not carried out for each stack operation adding something to the stack but done only at specific moments of the execution.

To be more precise, at block entrance (in run-time subroutine ENTRB), at procedure entrance (in subroutine ENTRPB), at the entrance of an implicit subroutine (in subroutine ENTRIS), and after reservation of the area for array elements (in subroutine RAD, for Real Array Declaration, and in subroutine TAV, for Take Array by Value) the new value

of the stack pointer, augmented by a certain margin, is checked against the end of the area that is available for the execution stack.

The margin mentioned above is computed by the compiler and incorporated in the object program after its last instruction, which is 'GOTO(:ENDRUN)'. It is the sum of four variables, the maximum stack movement within the elaboration of expressions outside implicit subroutines (max depth), the maximum stack movement of expressions within implicit subroutines (max depth isr), the maximum display length (max disp length) and the maximum stack growth in a procedure preceding the call of ENTRPB (max proc length). The execution of an expression

$$x + f1(f2)$$

(where f1 has a real parameter called by value and f2 is a parameterless real procedure) is guarded for the first time by subroutine ENTRPB, occurring in the object code for procedure f2. In the mean time the value of x, a new display for f1 and the actual parameter descriptor for f2 have been added to the stack, whereafter the execution of f2 by the value mechanism adds again a new display to the stack. Hence the unguarded stack growth could be as much as max depth + max proc length + max disp length for an expression outside implicate subroutines and max depth isr + max proc length + max disp length for an expression inside implicate subroutines. With the sum of all four quantities we are always on the save side.

Most likely this implementation of the guarding of stack overflow was designed at a late stage of the compiler design, for the four variables discussed above are obtained, in the part of the compiler called 'macro processor' (c.f. Section 5.3.4), by reconstructing the stack movements from the instructions produced by the translator scan of the compiler.

For the programs dealt with in the next chapter the margin varies from 4 (for Erathostenes) to 52 (for JAZZ164). At the start of program execution this margin is assigned to a variable *bcheck*, which then is used in all checks.

program	<i>bcheck</i>
erathostenes	4
pentomino	14
havy integral	26
lisp	32
QR	36
JAZZ	52

5.3 The structure of the compiler

5.3.1 Overview

The compiler consists of 4 parts:

1. a collection subroutines called ‘General–purpose procedures’. This collection contains a number of subroutines together constituting the lexical scan, and some other routines that are called from at least two of the other parts. Its length is 881 words in total.
2. the `prescan0` program. In `Prescan0` the program is scanned and a complete name list (identifier table) is constructed. 509 words.
3. the `prescan1` program. Collects, in the name list, all further data necessary for the translator scan and does the addressing of variables. 894 words.
4. the translator scan. Produces the object program in situ, ready for execution. 2965 words.

The complete length is 5249 words.

All three scans read the string of basic symbols that constitutes the program. In `Prescan0` this string is derived from the text on paper tape using the lexical scan subroutines. As a side effect the string is saved in store in an area called the text array. `Prescan1` and the translator scan read the string from the text array. All three scans build identifiers and numbers from letters and digits (including ‘.’ and ‘₁₀’) as contained in the string of basic symbols.

The compiler does a complete syntax check. For most errors, a simple back–on–the–rails procedure is applied to continue the compiling process, in order to find as many errors as possible. Syntactically incorrect are never executed, however.

The translator scan was written first and includes most of the syntax checks¹¹. It uses a recursive descent method for parsing and generates the object code. It assumes the availability of a block–structured symbol table, containing also some general data for each block or procedure. In the parsing procedures, however, no implementation details of the symbol table are present: the symbol table is structured as an abstract data structure with many functions and procedures to extract data from or to add new data to it. It

¹¹All syntax errors with error number from 300 (‘in an arithmetic expression an if–clause is not closed by **then**’) upto 401 (‘code body does not start with a quote symbol’) are reported by the translator scan.

was only after the completion of the design of the translator scan that the representation of the symbol table was chosen: then it was known what information it should contain.

Also the code generation as included in the parsing procedures is rather abstract: code is produced as macro's with or without a parameter, and in a special 'macro processor' these macro's are expanded to X8-instructions, possibly after applying some peep-hole optimization.

Both prescans were derived from the translator scan by leaving out all code for syntax checking and object program production and applying many other simplifications.

Prescan0 reads the source text from paper tape using the lexical scan subroutines. It produces a program listing on the line printer. Its main task is the construction of a complete symbol table. It mainly analyses and checks the block structure and the declarations and reports all errors found¹².

If serious errors are detected in the declaration structure (such as an identifier declared twice in the same block) the compiling process is disrupted at the end of Prescan0. This was not originally planned but showed necessary in the first weeks of practice. Consequently, in that case no further checks are carried out. It would have been better to do all context-free error checking in Prescan0 instead of the translator scan (in an ALGOL 60 compiler for Philips P1400, written in 1970 when I moved to the research laboratory of Philips in Eindhoven, I included all these checks in the first scan indeed).

The main task of Prescan1 is to add to the symbol table information about formal parameters that can not be derived from their specifications (if present at all!), but can be extracted from their use, such as the dimensions of array parameters or the number of arguments for a procedural parameter. Also the use of labels in inner blocks or as actual parameter is registered. Moreover, data is collected about the depth of inner block structure (determining the length of the display) and the depth of for-statement nesting (determining the number of locations to be reserved for the for-list variables of the block, cf. Section 5.2.3). There are only 4 (context-sensitive) error checks, among which the check on undeclared identifiers (after producing error message 204, the identifier is added to the symbol table as global identifier with the descriptor of an unspecified formal parameter).

The main task of Prescan1 as mentioned above requires a more detailed analysis of the program than needed in Prescan0.

¹²All syntax errors with error number from 100 ('in a parameter separator the colon is missing') upto 130 ('the value of a numeric label is non-integral or lies outside the integer capacity') are reported by Prescan0.

The ALGOL 60 version of the compiler, written and tested before its coding in ELAN, was designed with this coding in mind. Therefore, complex ALGOL 60 constructions were avoided. In general, the compiler procedures have a few parameters only, preferably value parameters, that in the ELAN coding can be transferred in a register.

5.3.2 The General-purpose procedures

In the first place the collection of general-purpose procedure contains the lexical scanner. Essentially it consists of three layers: subroutine 'next symbol', subroutine 'next basic symbol', and subroutine 'insymbol'.

The upper layer is function next symbol. It delivers (the internal representation of) the next basic symbol of the source program as function value and also stores it in variable 'last symbol'. Moreover, it sets two boolean variables, 'letter last symbol' and 'digit last symbol'. It also skips comment (starting with **comment** or following **end**) and assembles '*<letterstring>* : (' to (the internal representation of) ','. During Prescan0 it also takes care for buiding the text array.

The lower level, function insymbol, decides whether to read from the input source (Prescan0) or from the text array (Prescan1 and Translator scan). In the latter case it simply extracts the next symbol from the text array and exits. Otherwise, it has to assemble one or more characters from input to (the internal representation of) a basic symbol. It skips lay-out symbols (except the new-line symbols), assembles ':' and '=' to 92 (for ':=') and '_ ' 'i' ' ' 'f' to 94 (for **if**).

The only task of the intermediate level is bookkeeping of the line counter. When it meets a new-line symbol it increments the line counter by one and it calls insymbol for the next basic symbol. Only within strings the new-line symbols are delivered to the upper layer (and then to the parsing procedure calling next symbol).

Apart from the lexical scanner there are a.o.:

- routines for further analysis of basic symbols, like 'aritmetic operator last symbol', 'declarator last symbol' and 'specifier last symbol'. They deliver a boolean result and in case of an affirmative answer the latter two assemble, if appropriate, two or more basic symbol to one characteristic value for the declarator or specifier (e.g. **own integer array** to 41).
- a routine to assemble unsigned numbers, delivering its value in F and setting two boolean variables 'real number' and 'small' (the latter true for integer representations < 32768).

- a routine to assemble identifiers, delivering their representation at the end of the symbol table.
- a routine to look whether the identifier at the end of the symbol table is present elsewhere in that table. The parts of the table where to look for it depend on the context.
- routines to skip type declarations, value lists and specification lists (called from `Prescan1` and `Translator scan`).
- routines to administrate at block entrance and block exit some pointers governing the look-up process of identifiers in the symbol table.
- the procedure ‘skip rest of statement’, activated occasionally in the back-on-the-rails process after the detection of an error.

5.3.3 The translator scan

In the translator, having a total length of 2965 words, one can distinguish several parts:

- the recursive descent parser, checking the program and generating code in the form of macro’s (2073 words);
- the macro processor, generating X8 instructions in situ after application of some peep-hole optimizations (239 words);
- a set of symbol-table procedures, delivering properties of identifiers and blocks or adding new data, such as (code) addresses of procedures and labels (246 words);
- some tables for use by the macro processor (407 words).

As said before, I wrote the parser unaware of any theory about parsing, let alone of notions like LL(1) grammars or recursive descent parsers. As I was trained of programming in ALGOL 60 and almost thought in terms thereof, it was very natural to me to try to write a compiler by means of a procedure for each of the nonterminal symbols of the BNF grammar of ALGOL 60. In general such a procedure expects that the first symbol of the terminal production of that nonterminal symbol is already read and stored in variable ‘last symbol’ and leaves in turn the first symbol not belonging to it (the ‘follower’) in ‘last symbol’. Almost automatically I applied certain transformations where the grammar was not in suitable (i.e., LL(1)) form. We give an example:

ALGOL 60 rule:	$\langle \text{factor} \rangle ::= \langle \text{primary} \rangle \mid \langle \text{factor} \rangle \uparrow \langle \text{primary} \rangle$
LL(1) form:	$\langle \text{factor} \rangle ::= \langle \text{primary} \rangle \langle \text{next primary} \rangle$ $\langle \text{next primary} \rangle ::= \langle \text{empty} \rangle \mid \uparrow \langle \text{primary} \rangle \langle \text{next primary} \rangle$
ALGOL 60 code:	<pre> procedure Factor; begin Primary; Next primary end Factor; procedure Next primary; begin if last symbol = ttp then begin Macro(STACK); next symbol; Primary; Macro(TTP); Next primary end end Next primary; </pre>
ELAN code:	<pre> FACTOR: SUBC(:PRIMARY) NXT PRIM: S = last symbol U, S - 69, Z " last symbol = ttp? N, GOTOR(MC[-1]) A = 0 SUBC(:MACRO) " MACRO(STACK) SUBC(:NXT SBL) SUBC(:PRIMARY) A = 7 SUBC(:MACRO) " MACRO(TTP) GOTO(:NXT PRIM) </pre>

The LL(1) rules are nicely reflected in the ALGOL 60 code. We see moreover clearly how the parsing activities are mingled with code generation. The macro processor produces (the bit patterns corresponding to) instruction 'MC = F' out of macro STACK and instruction 'SUBC(:TTP)' out of macro TTP. The ELAN code follows the structure of the ALGOL 60 code conscientiously. The only adaptations are the fact that in NXT PRIM tail recursion is replaced by iteration and that the call of NXT PRIM at the end of FACTOR is eliminated by putting its code directly following the code of FACTOR.

A second example is given in the next table. In this example we took a decision that, for reasons to be explained below, could better have been taken differently.

ALGOL 60 rules:	$\langle \text{primary} \rangle ::= \langle \text{unsigned number} \rangle \mid \langle \text{variable} \rangle \mid$ $\langle \text{function designator} \rangle \mid (\langle \text{arithmetic expression} \rangle)$
LL(1) form:	$\langle \text{primary} \rangle ::= \langle \text{unsigned number} \rangle \mid$ $\langle \text{identifier} \rangle \langle \text{array extension} \rangle \langle \text{function extension} \rangle \mid$ $(\langle \text{arithmetic expression} \rangle)$ $\langle \text{array extension} \rangle ::= \langle \text{empty} \rangle \mid [\langle \text{subscript list} \rangle]$ $\langle \text{function extension} \rangle ::= \langle \text{empty} \rangle \mid$ $(\langle \text{actual parameter part} \rangle)$
ALGOL 60 code:	<pre> procedure Primary; begin integer n; if last symbol = open then begin next symbol; Arithexp; if last symbol = close then next symbol else ERRORMESSAGE(302) end else if digit last symbol then begin Unsigned number; Arithconstant end else if letter last symbol then begin n:= Identifier; Subscripted variable(n); Function designator(n); Arithname(n) end else begin ERRORMESSAGE(303); if last symbol = if \vee last symbol = plus \vee last symbol = minus then Arithexp end end Primary; procedure Subscripted variable(n); integer n; begin if Subscrvar(n) then begin Address description(n); if last symbol = colonequal then begin Macro(STACK); MACRO(STAA) end else Evaluation of(n) end end Subscripted variable; ... </pre>

Since in the ALGOL 60 rules a variable (either a simple variable or a subscripted variable) and a function designator both start with an identifier, in the LL(1) rules the identifier is factored out. In the ALGOL 60 routines we see this reflected by the call of function ‘Identifier’ followed by two procedure calls catering for the cases of a subscripted variable or a function designator. Identifier delivers a pointer in the symbol table pointing to the descriptor of the identifier.

The less fortunate decisions were taken in the ALGOL 60 procedures ‘Subscripted variable’ and ‘Function Designator’. Instead of inspecting ‘last symbol’ in order to see whether it is an opening parenthesis (in Function designator) or an opening square bracket (in Subscripted variable), the symbol table is consulted to see whether the identifier is the identifier of a function (in Function designator) or an array (in Subscripted variable). This does not only conflict with the idea of a one-symbol look-ahead parser. It makes it also impossible that both prescans and the translator scan base their parsing decisions in the same way, in the prescans the information contained in the symbol table not being necessarily available in time.

The above example also demonstrates some error checking and some back-on-the rails measures. In case of a missing closing parenthesis this fact is reported, and parsing is simply continued. It is possible that the preceding call of procedure Arithexp (short for Arithmetic expression) is prematurely ended because a symbol was found that could not belong to an arithmetic expression (e.g. a comma). In that case Primary just passes the problem upwards in the calling hierarchy.

If primary does not find a letter, a digit, or an opening parenthesis, it reports an error. In case of some frequently occurring errors it tries to resume parsing itself (as in ‘a + **if** x > y **then** x **else** y’ or ‘x↑ -2’), otherwise it again passes the problem upwards.

Of course I struggled at some places where ALGOL 60 is not an LL(1) language. Trying the impossible, this made some parts of the compiler more complicated than they otherwise would have been. The problem is that the context-free language

$$\{(n \ 1)^n = 2 \mid n > 0\} \cup \{(n \ \mathbf{true})^n \mid n > 0\}$$

is not LL, let alone LL(1), and this language is just a subset of the terminal productions of <Boolean expression>.

The most complex part is the analysis of actual parameters. According to the Revised Report we have the following rule:

```
<actual parameter> ::= <string> | <expression> | <array identifier> |
    <switch identifier> | <procedure identifier>
```


Therefore, actual parameters can be almost anything. A special case of `<expression>` is `<variable>`, which needs a special treatment because assignments to the corresponding formal parameter must be reckoned with. Since the procedure identifier in the call can be an formal parameter, we should also take into account that not always information about the nature and use of its arguments is available. Consequently, the analysis of an actual parameter is carried out almost bottom-up rather than top-down, without explicit expectation about what kind of actual parameter to expect. Only after the analysis of an actual parameter (under construction of the parameter descriptor and, if applicable, an implicit subroutine) there is a test whether it is compatible with what is known about the corresponding formal parameter.

The back-on-the-rails mechanism, touched on already above, was quite simple but rather effective. Most of the parsing procedures passed a serious problem, in which it was not clear how to continue, upwards in the calling hierarchy. There was one level, the compound-tail level, that was able to skip text in order to synchronize text and parsing again:

ALGOL 60 rule:	<code><compound tail> ::= <statement> end <statement> ; <compound tail></code>
LL(1) form	<code><compound tail> ::= <statement > <rest of compound tail> <rest of compound tail> ::= end ; <compound tail></code>
ALGOL 60 code:	<pre>procedure Compound tail; begin Statement; if last symbol \neq semicolon \wedge last symbol \neq end then begin ERRORMESSAGE(367); skip rest of statement(Statement) end; if last symbol = semicolon then begin next symbol; Compound tail end end Compound tail;</pre>

Three notes:

1. Compound tail is the only procedure that does not read beyond the last symbol of the construction (i.e. the **end** symbol), because that symbol could be the very last symbol of the input string!
2. Procedure 'skip rest of statement' is the only compiler procedure with a procedure parameter. It is called from all three scans, each with its own procedure 'Statement'.
3. Procedure 'skip rest of statement' skips text until it meets a semicolon or an **end**

symbol, but does so in an intelligent way: if it meets a **do**, a **goto**, a **for**, or a **begin** it calls its parameter in order to parse a statement before continuing skipping. Also when it meets an opening string quote it skips the string until the corresponding closing string quote before continuing normal skipping.

5.3.4 The macro processor

In the translator scan the object program is generated in the form of a sequence of macros. It is the macro processor that generates the real X8 instructions. In two tables it has for each macro the instructions and some properties at its disposal. Macros can have parameters. For practical reasons the number of parameters is limited to one and the number of instructions per macro to three. In some cases, therefore, a macro was split into two macros that always follow one another. An example is the macro pair DPTR(display level) and INCRB(top of display), leading to the instructions:

```

S = D                                " )
A = - (display level - 2), Z        " ) DPTR(display level)
SUB0(:DPTR)                          " )

B + (top of display)                " INCRB(top of display)

```

which we met before in Section 5.2.4.

A special task of the macro processor is to carry out some peep-hole optimizations. We give an example. For the arithmetic expression 'x + 1', translation scan procedure 'Arithexp' generates the macro sequence:

```

TRV(x)    " Take Real Value of x
STACK
TSIC(1)   " Take Small Integer Constant 1
ADD

```

The last three macros are optimized to: ADDSIC(1)

```

ADDSIC(1) " Add Small Integer Constant 1

```

leading to the instruction:

F + 1 " Add 1 to F

For this purpose the macro processor can be in several states, indicating which macros it has in stock that are candidates for optimization. Whenever the translator scan refers to the instruction counter of the object program (pointing to the place where the next instructions have to be placed), first the stock of the macro processor is emptied by generating the corresponding instructions.

The macro parameter of macros like TRV, Take Real Value, is a pointer into the symbol table, to the descriptor of the identifier of a variable or a formal parameter. For such macros the macro processor converts this pointer to the (static or dynamic) address of the variable and adapts the addressing variant of the instruction when applicable.

Another task of the macro processor is to register the stack movement caused by the execution of a macro. The purpose of this is to determine the number that is used in guarding stack overflow during program execution as discussed in Section 5.2.8. There are several reasons to do this in the macro processor. First of all this makes the code of the translator scan more transparent. The stack movement is, furthermore, influenced by the peep-hole optimization of the macro processor, as can be seen in the example above. Finally it could be the case that the monitoring of the stack movements was planned at a stage where large parts of the translator scan were already designed. The price paid is that the macro processor, to which the structure of the source program is not available, has to do some reconstruction activities with the help of a table of macro properties.

5.3.5 The symbol table

We discuss here the structure of the symbol table. For each block (including those of procedure declarations) there is a segment of words in store containing:

- a heading, giving some general data for the block,
- a section for the formal identifiers of the block, if any,
- a section for the local identifiers of the block, possibly interspersed with the segments of inner blocks.

The heading contains, a.o., pointers to the block heads of the smallest enclosing block and of the textually following block, pointers to the first formal identifier and to the first local one, a pointer to one of the descriptors of for variables, the block number (i.e. the number of enclosing blocks), and several other numbers. It consists of 5 words.

All characters of an identifier are stored in the symbol table, 4 characters per word, requiring $4 * 6$, i.e. 24 bits. These words are stored negatively. The descriptor of the

identifiers occupies one, two or three words, immediately following the character words, and are stored positively. The first descriptor word contains a code of 7 bits. Three of these give its type (real, integer, Boolean, string, arithmetic, non-designational, designational, unknown), one bit discriminates formal and non-formal, and three bits determine its character (simple variable, simple own variable, array or switch, own array, procedure, function). The first descriptor word contains moreover an 18-bits address and one bit indicating for non-formals whether the address is static or dynamic, whereas for formal identifiers that bit indicates whether the formal is called by value or by name.

In case of a formal identifier, an array, a procedure, or a function there is a second descriptor word, containing more information, giving the dimension for arrays and the number of parameters for procedures and functions. For non-formal function identifiers there is a third descriptor word containing the descriptor of a simple local variable, used to record the function value.

Apart from some exceptions to be discussed below, the identifiers of a block are stored successively. The identifier look-up procedure is rather simple. As soon as a mismatch is found, the rest of the (negative) character words and the positive descriptor words are skipped and the next (negative) character words are examined.

In general, however, the symbol table contains 'jump instructions' indicating that the search has to be continued elsewhere in the symbol table. Jumps are coded by negative words in which not all first three bits are one. There are two different kinds of jumps:

- a jump over the segment of an inner block, and
- a jump at the end of the locals or formals of a block.

The former kind of jump occurs when among the declarations of a block a (type) procedure is found. After the procedure's descriptor there is a jump to the next local identifier of the block. Also after the segment for an inner block among the statements new local identifiers of labels can be present.

The latter kind of jumps occurs at the end of the 'local list', leading to the 'formal list' (which is empty for an inner block or for a procedure without parameters) and at the end of the formal list, leading to the local list of the enclosing block (which list, of course, might also be empty).

Since after `Prescan1` all identifiers are present in the symbol table, the look-up procedure always ends with a hit.

The translator scan is completely ignorant of the structure of the symbol table: the latter is an abstract data type. The symbol table section of the translator scan contains some 40 functions to interrogate the symbol table about the properties of an identifier and a

few procedures to fill in some new information (e.g. the program address of a label or procedure).

5.3.6 Prescan1

The main task of Prescan1 is to collect data about the use of identifiers and store these in the symbol table for use by the translator scan.

In the first place it adds data about the use of formal parameters. According to the definition of ALGOL 60, it is not necessary to give a specification of formal parameters, except for those that are called by value. But even if a formal parameter is specified, this specification is often far from complete. Specification '**boolean array** b' does not give any information about b's dimension. From statement '**b**[i,j]:= **true**' it follows that b is a boolean array of dimension 2.

All information gathered in Prescan1 is reported to the symbol table by means of procedure calls. Unknown facts are added, facts already registered in the symbol table or conflicting with previously stored information are simply neglected.

Prescan1 adds an undeclared identifier at the end of the symbol table as an unspecified formal (after producing an error message). It is treated as all other formals, so every applied occurrence of that identifier leads to information on its intended nature, which in the translator scan is used to check its consistent use.

Some other facts are registered about the use of identifiers. For formal parameters occurring as left part of an assignment statement that fact is registered. It is used in the translator scan to generate a call of CLPN (Call Left Part by Name) as described in Section 5.2.4. Any use of a label identifier other than in '**goto** <label identifier>' local to the block in which that label has its defining occurrence leads to a mark in the symbol table. This results at run time in the creation of a 'label variable' in the block cell in which the label's program address and the block's display pointer and block number are stored.

For each block also the maximal nesting depth of for statements is registered.

A final task of Prescan1 is to assign addresses to all local variables and arrays.

Prescan1 was written after completion of the translator scan. It was obtained by leaving out all code generation and all error messages, while adding the information gathering. Moreover, where possible the syntax analysis was simplified. Typical examples are routines 'Simple arithexp' and 'Subscripted variable', which we present in their ALGOL 60

version. The calls of 'Arithmetic', 'Subscvar' and 'List length' herein potentially add new information to the symbol table.

```

procedure Simple arithexp;
begin integer n;
    if last symbol = plus  $\vee$  last symbol = minus
    then
next0: next symbol;
    if last symbol = open
    then begin next symbol; Arithexp;
        if last symbol = close then next symbol
    end
    else
    if digit last symbol then unsigned number
    else
    if letter last symbol
    then begin n:= Identifier; Arithmetic(n);
        Subscripted variable(n); Function designator(n)
    end
    else
    if last symbol = if then Arithexp;
    if arithoperator last symbol then goto next0
end Simple Arithexp;

procedure Subscripted variable(n); integer n;
begin if last symbol = sub
    then begin Subscvar(n)
        dimension:= Subscrlist; List length(n)
    end
end Subscripted variable;

```

5.3.7 Prescan0

Prescan0 reads the ALGOL 60 program from external source and constructs a first version of the symbol table. As a side effect the basic symbols that constitute the program are stored in a text array for use in Prescan1 and the translator scan. All lay-out characters except the new-line characters are thereby removed, as are all comments. Moreover the

text is listed, augmented with line numbers.

For the construction of the symbol table the block structure of the program and the declarations are carefully analyzed. Statements are almost completely skipped. The only points of interest are the occurrence of labels (that have to be added to the symbol table) and the basic symbols **for** (for administrating the maximal nesting of for-statements in the block), **begin** (possibly the start of an inner block) and string quote open (in order to be able to skip strings separately).

The construction of the symbol table is carried out with care. For each identifier to be added it is checked that it really new, i.e. that it does not have a defining occurrence for the same block in the preceding text. If there is, however, such an occurrence, this is considered as a serious error and a reason to disrupt the compilation process at the end of Prescan0.

The code of Prescan0 in its ALGOL 60 version is a procedure 'prescan0' with 11 local procedures and a body. The local procedures are named 'Program', 'Block', 'Compound Tail', 'Declaration list', 'Statement', 'Label declaration', 'Int lab declaration', 'Begin statement', 'Store numerical constant', 'Process identifier', and 'Identifier'. We give here the code of 'Statement' ($6 \times d19$ being the descriptor value for a label):

```

procedure Statement;
begin integer n, lfc;
    lfc:= local for count; character:= 6 × d19;
next: if letter last symbol
    then begin read identifier;
        if last symbol = colon
            then begin n:= Process identifier;
                Label declaration(n);
                goto next
            end
        end
    else if digit last symbol
    then begin unsigned number;
        if last symbol = colon
            then begin Int lab declaration;
                goto next
            end
        else Store numerical constant
        end
    else if last symbol = for
    then begin local for count:= local for count + 1;
        if local for count > max for count
            then max for count:= local for count
        end
    else if last symbol = begin
    then begin Begin statement; next symbol; goto next end
    else if last symbol = quote then skip string;
    if last symbol ≠ semicolon ∧ last symbol ≠ end
    then begin next symbol; goto next end;
    local for count:= lfc
end Statement;

```

Prescan0 was written after, and almost indepently of Prescan1.

5.3.8 Results of some measurements

For one of the programs (JAZZ164) discussed in the next chapter we present here results of some measurements on compiler execution. The first table gives the number of paper-tape punchings read and the number of basic symbols constructed from these by the lexical-scan procedures¹³.

number of paper-tape punchings	9 477
number of basic symbols	3 479

Next we give the number of instructions executed and the time spent in several parts of the compiler. The data for the lexical scan are time and number of instructions used for the production of the basic symbols. During Prescan0 they are constructed (by reading paper tape), stored and printed, during Prescan1 and the translation scan they are taken from store. The assemblage of identifiers and numbers from their constituent basic symbols is carried out in the scan themselves and done anew in each of the scans. The figures for the translation scan include the activities of the macro processor.

	time (msec)	instructions
Prescan0	4 189	1 057 553
of which lexical scan	3 756	945 074
look up	57	13 331
Prescan1	1 741	453 949
of which lexical scan	412	113 505
look up	550	135 137
Translation scan	2 989	758 263
of which lexical scan	412	113 505
look up	536	132 160
macro processor	871	209 894
total compilation	8 919	2 269 765

¹³Both the number of punchings and the number of basic symbols differ from those reported in [8]. There are three causes for these differences:

1. the program text for the X8-version of the program differs from the original version: in the X8 version the own arrays had to be replaced by global arrays, a declaration for procedure SUM had to be added, and the procedure calls of FLOT needed a third parameter;
2. in the X8-version symbols between symbol **end** and the next **end**, **else** or semicolon are already skipped in the lexical scan and therefore not counted as basic symbols;
3. new-line symbols are in the X8-version counted as basic symbols, since they are used in the line-number administration.

If we express the timing figures as percentages of the total compilation time we get the following results:

	time (%)
Prescan0	47.0
of which lexical scan	42.1
look up	0.6
Prescan1	19.5
of which lexical scan	4.6
look up	6.2
Translation scan	33.5
of which lexical scan	4.6
look up	6.0
macro processor	9.8

We see that the lexical scan activities in Prescan0, i.e. reading and printing the text, the isolation of basic symbols, and their storage for retrieval in the next two scans, costs already almost half of the compilation time. The analysis of the text in Prescan1 costs less than 9%, including the incorporation of the data gathered into the symbol table. In the translation scan about 13% of total time is enough to do the scanning, including all checks and the generation of the macros. The elaboration of these macros into X8 instructions costs almost 10%, hardly less than the scanning! The moral is that it is important to do the lexical analysis as efficient as possible, whereas parsing at the context-free level is done almost for free.

Chapter 6

Comparison of the two ALGOL 60 systems

In this chapter we compare the two ALGOL 60 systems that were described in the previous chapters.

We start by comparing some of the characteristics of the two machines, the EL X1 from 1958 and its successor, the EL X8 from 1965.

Next we compare the ALGOL 60 implementations for these two machines with respect to both the compilers and the object programs produced by them.

Thereafter we present and analyse figures that were collected by compiling and executing six ALGOL 60 programs on both the ALGOL 60 system for the EL X1 and that for the ELX8, using emulators for these machines written in Pascal.

6.1 Comparison of the EL X1 and the EL X8

In its time the EL X1 was a quite modern machine: fully transistorized, core store, an index register, and an interrupt system for I/O. The central processor fitted in a large writingdesk. Besides the conventional type writer and tape-punch peripherals also punch-card equipment and magnetic tapes could be connected (but the MC never bought these). It was aimed mainly at use in administration. It had a short word length of 27 bits, therefore a small integer range. It had no floating-point provision and no support at all for the implementation of programming languages. Nevertheless it was a nice machine

and the MC used it heavily for scientific computation.

The successor of the El X1, the El X8, was about 12 times as fast as the X1 and (almost) upwards compatible with it. It had, as we saw, a number of extensions directed to the implementation of ALGOL 60 (and other programming languages): a floating-point register, stack instructions, several new addressing variants, a provision for fast context switches, and a provision for parameter handling. As a result execution of ALGOL 60 programs ran a factor of 40 to 60 faster than on the X1. The minimal machine was housed in 5 cabinets plus a console (mainly for use by technical staff and software developers). All peripherals were on separate tables. The hardware used integration at the gate level.

The X8 was lacking memory protection and a separate monitor mode. That made it unsuited to run multiple machine-code programs on it. It was, however, possible to run ALGOL 60 programs in multiprogramming: the THE system of Dijkstra et al. [6] is a famous example.

Some data:

	X1	X8
first delivery	1958	1965
integration level	none	gate level
word length	27	27
registers	A,S (27 bits) B (15 bits)	A,S (27 bits) B (27 bits) F (54 bits)
instruction counter	T (15 bits)	T (18 bits)
index registers	B	A,S,B,F,T,M[63]
store cycle	32 μ sec	2.5 μ sec
store in units of	4 K words	16 K words
max core size	32 K words	256 K words
A = M[1000]	64 μ sec	5 μ sec
MULS(M[1000])	500 μ sec	max 40.00 μ sec
F + M[1000]		max 18.75 μ sec
F \times M[1000]		max 68.75 μ sec
F / M[1000]		max 68.75 μ sec

6.2 Comparison of the ALGOL 60 implementations

It is difficult to underestimate the achievement made by Dijkstra en Zonneveld when they implemented ALGOL 60 on the EL X1. Almost everything was new: the language, the machine, run-time organization for blocks and (recursive) procedures, and compilation. ALGOL 60 itself was still under development, the X1 was a computer not really fit to implement such a language, there were no existing implementations to learn from, and the compiler and all of its working space had to be accommodated in a store of 4 K words only. The resulting compiler was 2 K instructions short.

For the crew constructing the ALGOL 60 implementation the situation was quite different. The language was well known, there was experience with its implementation, the X8 had provisions for the implementation of higher order programming languages (such as stack instructions including a subroutine call putting its link on top of stack), and the store size of the X8 was at least 16 K words. Nevertheless the design of the object-program structure, making optimal use of the new machine features, was quite an effort, whereas the structure of the compiler was completely new and different from that for the X1.

We come back to some of these points in the next sections.

6.2.1 Comparison of the object programs

The essence of the object code for the X1 was the use of pseudo instructions, calls to subroutines in a complex of subroutines supporting execution. It is comparable with a P-code, i.e. the instruction code of an emulated stack computer.

The object programs of the X8, on the other hand, used in principle the instructions of the X8 itself; only for more complex tasks, such as indexing in arrays or block and procedure entrance, subroutine calls in a complex of subroutines were used.

The following example, giving the object codes for statement 'x:= x + y', shows this difference clearly.

X1:	B = @x	" load address of x in register B	36 μ sec
	SUB1(:TRAS)	" Take Real Address Static	500 μ sec
	B = @x	" load address of x in B	36 μ sec
	SUB1(:TRRS)	" Take Real Result Static	908 μ sec
	B = @y	" load address of y in B	36 μ sec
	SUB2(:ADRS)	" ADd Real Static	2868 μ sec
	SUB1(:ST)	" STore	1400 μ sec
X8:	F = M[@x]	" load value of x in register F	7.50 μ sec
	F + M[@y]	" add value of y to F	11.25 μ sec
	M[@x] = F	" store F in x	7.50 μ sec

Note that the X8 object code uses a register for buiding up results where the X1 code uses the (top of) stack for that purpose. The X1 version puts the address of the left-hand side of the assignment statement, together with type information, on top of stack before evaluating the right-hand side expression. The result of the latter, again stored on top of stack, includes also type information. Subroutine ST can, therefore, carry out the proper assignment. In the X8 code, on the other hand, the address information is moved beyond that expression and used in a store instruction that is selected on the basis of its type.

Execution times (for $x = 3.14$ and $y = 0.1$) are 5784 μ sec for the X1 and 26.25 μ sec for the X8. The addition itself costs 2868 μ sec for the X1¹ and 11.25 μ sec for the X8. Here we see in isolation the effect of having floating-point operations in hardware!

The complex of the X1, excluding the subroutines for input and output and those for the standard functions *sqrt*, *sin*, *cos*, *ln*, and *exp* is 1107 instructions long. The complex of the X8 counts 627 instructions only.

6.2.2 Comparison of the compilers

The ALGOL compiler for the X1 has two passes. Both passes read the ALGOL 60 program from source in a cyclic process which proceeds from delimiter to delimiter. The first pass collects the identifiers of procedures, switches and labels from their defining occurrences. The second pass generates and punches the object code. It contains a separate piece of program for each delimiter. The context is characterized by 6 Boolean values. A stack

¹In fact ADRS is a combination of TRRS and ADD. Execution of TRRS and ST imply changes from variable representation to stack representation and vice versa. The execution times of these changes form part of the execution times for TRRS and ST.

is used to store operators (like ‘+’ and ‘∨’, but also ‘:=’, ‘(’, and ‘if’) and parts of the context values at a context switch.

In this way a minimum of information needs to be stored by the compiler. In the compilation of statement ‘ $x := x + y$ ’ first delimiter ‘:=’ is encountered. It is stacked with low priority and the first two instructions of the object code given in the previous section are generated. Next delimiter ‘+’ is met. It is stacked with somewhat higher priority and the next two object instructions are generated. Finally the delimiter following the assignment statement is found and the last three instructions are generated on the basis of the two operators found in the stack. Execution, however, of the object code takes 12 stack positions.

The X8 compiler has three passes. Each reads the source program anew, but the last two passes do so from a text array in which the first pass stores the complete string of basic symbols. All three scans have the structure of a recursive descent parser. In the first two passes a complete symbol table is built. The third scan produces the object code in situ.

The compilation of statement ‘ $x := x + y$ ’ proceeds as follows. Compiler subroutine² *statement* (with its link on the stack) reads identifier ‘x’ and delimiter ‘:=’. It calls subroutine *assignment statement* with in S a pointer to the descriptor of ‘x’. There the type of ‘x’ is stacked and subroutine *real assignment* is called. This subroutine stores the descriptor pointer (still in register S) in the stack, reads identifier ‘x’ and delimiter ‘+’, generates the instruction for loading the value of ‘x’ to register F, and calls subroutine *rest of arithmetic expression*³. The latter reads the rest of the expression and produces the second instruction of the object code. Then subroutine *real assignment* unstacks the descriptor pointer and generates the third instruction, which completes the compilation of the assignment statement. It uses all together 17 stack positions.

²We describe here the ELAN code of the compiler.

³Subroutine *real assignment* cannot simply call subroutine *arithmetic expression* to compile the right-hand side of the expression: it has to cater for statements like ‘ $x := y := x + 1$ ’. An identifier following delimiter ‘:=’ can therefore be a next left part.

6.3 Results of measurements

First we recapitulate some figures that were given in Chapter 3 and Chapter 5. Thereafter we present figures that were collected by compiling and executing six ALGOL 60 programs on both the ALGOL 60 system for the EL X1 and that for the ELX8, using emulators for these machines written in Pascal.

These figures should be looked at with all proper reserves.

For the X1, the waiting times for peripheral devices are not taken into account. Whenever an I/O order for a device is given, it is assumed that that device accepts the order immediately and generates an interrupt signal instantaneously. The interrupt programs, however, are carried out correctly. Another deviation from the situation as it existed at the Mathematical Centre is the fact that an X1 is emulated with 24 K core store (instead of 12 K). This influences the compile time in two respects. In the first place, memory management for a number of lists used during compilation is simpler. Secondly, at the end of the compilation process the ‘free’ store is filled with octal value ‘777 777 777’ in order to make program execution reproducible. In a 24 K store this costs the execution of an additional 24 K instructions compared to a 12 K store. The 24 K core store is large enough to execute both the sieve of Erathostenes and the lisp interpreter (see Section 6.2.1) on the X1, whereas the other programs execute in a 12 K machine too. A further difference with the MC X1 is that the list of identifiers of built-in procedures and functions in the emulator is smaller and that less MCP’s are processed during the loading phase of the compiler.

In the X8 case I/O is handled by Charon. Input is read by Charon in portions (in the PICO case of 32 heptads) and output is presented to Charon in portions (in PICO of 150 lineprinter symbols). Again it is assumed that Charon handles commands instantaneously. Where in reality Charon uses the main store on the basis of cycle stealing, this fact is omitted in the emulator. Output for the lineprinter is packed in the form of 8-bit ASCII characters with three characters per word instead of four 6-bits characters in the special lineprinter code. Finally, the execution time for floating-point operations is taken from a model for the hardware as discussed in Section 4.6.

Nevertheless we believe that the figures given in the next sections are fully representative for the two ALGOL 60 implementations.

6.3.1 Summary of results of Chapter 3 and Chapter 5

In the following table we summarize some figures that were given in Chapter 3 (on the ALGOL 60 implementation for the X1) and in Chapter 5 (on the X8 ALGOL 60 implementation). We present for a number of basic constructions the number of instructions generated, the number of instructions executed, and the execution time. Moreover, in the last column, we give the ratio of the X1 and the X8 execution times.

statement	X1			X8			X1/X8 time
	code	executed	time	code	executed	time	
<code>i:= i + 1</code>	7	60	3 284	5	5	20.00	164
<code>x:= x + y</code>	7	110	5 784	3	3	26.25	220
<code>x:= x × y</code>	7	101	6 676	3	3	77.50	86
<code>x:= x / y</code>	7	105	6 946	3	3	78.75	88
<code>A[i]:= A[i] + x</code>	14	216	11 532	10	41	201.25	57
<code>I[i]:= 0</code>	8	81	4 416	6	27	120.00	37
<code>B[i]:= true</code>	8	81	4 416	6	39	192.50	23
for <code>i:= 1</code> step 1 until <code>n</code> do	22	179 + n*143	10 044 + n*7 612	21	15 + n*15	66.25 + n*83.75	91
<code>p(x)</code>	5	78	4 540	2	45	182.50	25
<code>p(x+0.1)</code>	10	78	4 540	7	46	185.00	25
<code>p1(x)</code>	5	163	8 912	2	47	207.50	43
<code>p1(x+0.1)</code>	10	320	17 824	7	61	278.75	64
<code>p2(x)</code>	5	230	12 476	2	68	307.50	41
<code>sqrt</code>	1	47	4 453	1	45	343.75	13
<code>sin</code>	1	229	24 557	1	32	656.25	37
<code>cos</code>	1	298	28 077	1	33	603.75	46
<code>exp</code>	1	232	24 195	1	76	985.00	26
<code>ln</code>	1	369	28 010	1	58	700.00	36
<code>arctan</code>	5	1500	106 523	1	61	785.19	135

The ratio X1/X8-times for these constructions is varying from 13 (for ‘sqrt’) to 220 (for ‘x:= x + y’).

The proportions for ‘x:= x × y’ and ‘x:= x / y’ show most clearly the effect of interpretation of a stack machine for the X1 versus direct execution of X8 instructions. For ‘i:= i + 1’ and ‘x:= x + 1’ the availability of floating-point hardware in the X8 leads to

an additional factor⁴. The large factor for ‘arctan’ can be attributed to a different and more efficient algorithm.

The small factor for ‘sqrt’ is mainly caused by the fact that the whole calculation but the last Newton iteration is carried out in 27–bits arithmetic. The factors for subscripted–variable handling are affected by the use of subroutines also in the case of the X8. Moreover does the X8 index subroutine a check of all index values against the corresponding lower and upper bounds. The ratio for the handling of Boolean arrays is specially low by the fact that in the X8 27 array elements of Boolean type are packed in one X8 word.

It is evident that with such a strong variation in the X1/X8 speed up for elementary constructions the execution times of complete programs will show a variation in execution–time speed up too, depending on the specific mix of elementary constructions in them. This is affirmed by the figures given in Section 6.3.5.

6.3.2 An estimation of the X1 to X8 acceleration factor

In 1969 I left the Mathematical Center for a job at the computer center of Philips Research Laboratories. It had an X8 at its disposal, and after a while also a Philips computer, a P1400, was installed with a time–sharing system called MDS. It contained an ALGOL 60 implementation that was incompatible with the X8 system and, moreover, was rather slow. Therefore it was decided to develop an ALGOL 60 system for the P1000 series, fully compatible with that of the X8. It took me less than one year to complete it.

Before embarking into that project I tried to find out whether the X8 system contained serious efficiency shortcomings. We devised a small system change by which periodically, through a clock interrupt, the current instruction counter and the binary code of five instructions around that position were recorded. These were collected during two days and analysed. They gave enough information to construct the following table:

⁴For multiplications and divisions the absence of floating–point hardware in the X1 plays a less important role: the multiplication of two 52–bit mantissas can be carried out by three 27–bits multiplications of 500 μ sec each.

DYNAMIC FREQUENCIES IN ALGOL ON THE X8

array indexing	23
arithmetic expressions	14
block/procedure entrance/exit	12
compiler	12
I/O	10
sqrt/exp/ln/sin/cos/arctan	10
for clauses	4
Boolean expressions	3
assignments	2
all remaining constructions	3
idle	7

The figures from this table reassured us: at least 24% of the time was used for real calculations (in arithmetic expressions and standard functions). Considering compile time and I/O time as unavoidable (in an experimental environment a program is seldom executed without some changes) the only point that needed some improvement was array indexing, although a reduction of indexing time by a factor of two would not lead to a substantial system-throughput increase. In the ALGOL 60 system for the P1400 array indexing was made more efficient for one- and two-dimensional arrays.

We will now try to apply this table in a quite other direction: can we learn from it something about the relation between X1 and X8 execution times? If we leave out the idle and compile times and neglect the contribution of I/O (since a large proportion of I/O is used in the compilation process, especially for syntactically incorrect programs that have no execution at all) we get the following figures for program execution:

construct	X8 execution	X1/X8 factor	X1 execution
block/procedure indexing	0.169 0.324	40 32	6.8 10.7
sqrt/exp/sin/ldots expressions	0.141 0.239	40 150	5.6 35.9
assignments for clauses	0.028 0.056	40 91	1.1 5.1
other constructs	0.042	12	0.5
sum	0.999		65.7

Here we tried to compute the execution time on the X1 for the program mix of our experiments using the X1/X8 factors that were gathered in the foregoing (cf. Sections 6.2.1 and 6.3). The value of some of these factors is quite arbitrary: we do not know the relative frequencies in which the standard functions *sqrt* etc. are used, we have no information what percentage of the expressions are of integer type, or which part of the assignments are assignments to formal variables. Nevertheless, rough estimations as it might be, we obtain a reasonable acceleration factor X8/X1 of 66.

6.3.3 The programs

For our measurements we used the following six ALGOL 60 programs:

- Havie integrator: integrates numerically three functions using a variant of Romberg integration [15]
- QR eigenvalues: computes the eigenvalues of a symmetric 5 * 5 matrix using a method of Housholder [4]
- JAZZ164: computes planetary trajectories using a Runge–Kutta method [19]
- Erathostenes: counts the number of primes below 10 000 using the sieve method of Erathostenes
- pentomino: computes the first 7 solutions of the standard plane pentomino puzzle using backtracking
- lisp interpreter: solves, interpreting the program written in LISP 1.5 derived in [14], the 3 missionaries and 3 cannibals crossing a river puzzle using a simple LISP 1.5 interpreter written in ALGOL 60

The first three programs are typical for scientific computations (perhaps missing a program solving linear equations). The other three programs work with integer and Boolean types only⁵.

The programs are listed in Appendix A. In fact the (emulated) X8 lineprinter output of running these programs is given there, including program listing, execution output and an execution profile.

⁵The lisp interpreter declares some objects of type real, but these are not used in solving the puzzle.

6.3.4 Compiling

Here we present some figures for the compilation process of the six programs for both machines: the length of the object program, the number of instructions executed during compilation, the compile time, the average instruction time, and the average number of compiler instructions executed to produce one object-code instruction. The compile time is given in seconds, the average instruction time in μ seconds.

machine	program	length	instructions	time	time/instr	instr/length
X1	Havie integrator	672	435 096	23.8	54.75	647
X8		609	787 624	3.08	3.92	1293
X1	QR eigenvalues	1354	607 264	32.7	53.98	448
X8		1211	1 240 976	4.87	3.92	1024
X1	JAZZ164	2538	1 145 895	61.9	54.07	451
X8		2497	2 270 261	8.92	3.93	909
X1	Erathostenes	170	156 361	9.1	58.21	920
X8		208	139 858	.6	3.96	672
X1	pentomino	973	505 350	27.5	54.43	519
X8		1019	960 488	3.75	3.90	946
X1	lisp interpreter	5290	3 159 178	167.5	53.05	597
X8		5128	7 497 415	29.1	3.88	1462

We can make the following observations.

- The object-code length for the two systems is of the same order of magnitude: their ratio varies from 0.82 to 1.12 (the X8 object code including the instructions for administration of line numbers).
- The number of instructions executed during compilation is roughly proportional to the object-code length; X8 to X1 ratio varies from 0.89 to 2.05. In general, compilation on the X8 needs twice as much instructions. This has several causes: program listing during Prescan0 (remind that the lexical scan, which includes listing, already takes 42% of the compile time), the complete check of the syntax, and the greater amount of information about the program to be accumulated.
- The average time per compiler instruction is, as to be expected, rather constant in both systems: about 55 μ seconds for the X1 and 3.9 μ seconds for the X8, reflecting a hardware speed-up of a factor 12. Since the X8 compilation process needs in

general twice as much instructions, compilation time is speeded up only by a factor of 6.

The fact that in the table the sieve-of Erathostheness program is a bit exceptional comes from the fact that it is a 17-line program of 7 statements only, three of which being for-statements (one of these a nested for-statement).

6.3.5 Program execution

For the execution of the six programs we list in the table below the number of instructions executed, the execution time, the average instruction execution time. The last column gives the ratio between the X1 and the X8 execution times.

machine	program	instructions	time	time/instr	X1-time/X8-time
X1	Havie integrator	59 374 267	4 380.5	73.78	45
X8		9 788 838	96.7	9.88	
X1	QR eigenvalues	280 120	15.8	56.56	55
X8		51 363	0.3	5.55	
X1	JAZZ164	9 450 329	557.3	58.97	54
X8		1 705 131	10.4	6.08	
X1	Erathostenes	4 997 305	268.2	53.68	42
X8		1 230 657	6.3	5.16	
X1	pentomino	63 812 331	3 471.2	54.40	67
X8		10 813 999	51.4	4.76	
X1	lisp interpreter	422 069 848	23 505.5	55.69	37
X8		135 622 946	636.7	4.69	

We see here a speed-up of by a factor between 42 and 67, much more than the factor 12 of the hardware speed-up. The precise factor depends on the program. Below we try, for each of the programs, to mention some of its characteristics contributing to its execution speed-up factor.

The largest speed-up factor, of 67, has program 'pentomino'. In the heart of this program, in lines 48 to 78⁶, all unused stones are successively examined in all their orientations as possible candidates for the next stone to be laid down in the rectangle of 6 by 10. The ALGOL 60 constructions heavily used are for-statements (25% of the X8 execution time),

⁶See Appendix B.

Boolean-array access (20% execution time), integer additions, and integer array access, constructions we have seen to speed up (in certain contexts) by factors 91, 23, 164 and 37. Procedure calls and parameters play no role of any importance. In this heart of the program 95% of the execution time is spent.

The ‘lisp interpreter’, on the other hand, consists of a large collection of very short procedures, e.g. for the operations ‘car’ and ‘cdr’ (see lines 124 to 134)⁷. Sometimes actual parameters are structured, leading to implicit subroutines and context switching. The procedure and parameter mechanism has, as we have seen, a moderate acceleration factor.

The small factor of the sieve of ‘Erathostenes’ can be fully attributed to the use of the Boolean array ‘prime’. Here the statements of lines 6 and 11 are responsible for 55% and the test of line 16 for 12% of X8 execution time. The acceleration factor of 23 for Boolean-array access is compensated partly by that for the for-clauses of lines 5, 10, and 15, catering for 33% of execution time. This mix results in a factor 42 for the program as a whole.

The execution time of the ‘QR eigenvalues’ program is more scattered over the program. Important is the statement in lines 60 and 61, containing two calls of the function ‘sum’ to compute the inner product of a matrix row and column. One of the actual parameters of ‘sum’ is complex, leading to implicit subroutines and context switches. The time spent in ‘sum’ caters for 13% of the total execution time. It is also used in line 52, and including the time spent in ‘sum’ these lines are responsible for 20% of total execution time.

Hence a favourable factor of 55.

Also ‘Jazz164’, which uses a Runge–Kutta method to integrate multiple coupled differential equations (here of planetary orbits) numerically, uses Jensen’s device intensively. As is the case in the QR eigenvalues, (real) array access plays also an important role.

⁷This interpreter was written for educational purposes. Separation of concerns and a clear decomposition were the major premises, efficiency was not (although we tried to minimize garbage production). Hence we hid in the major routines, like *assoc*, *pairlis*, *apply*, *eval*, *evcon*, and *evalis* the representation of lisp objects.

6.4 The influence of line–number book–keeping

In Section 5.2.7. we discussed the book–keeping of line numbers in the X8 system at run time for the purpose of easier error location in the case of a run–time error. It was possible, however, to suppress this book–keeping, thereby gaining in execution speed. The savings in program length varied from 15 to 27% in our six sample programs, whereas the speed savings varied from 0.5 to 6.5%. We now go into the question why these saving vary so much as they do.

Savings in execution time of more than 6% were measured for two of the programs: ‘pentomino’ and ‘lisp interpreter’. Both programs are characterized by relatively short program lines.

For ‘pentomino’ the object–program length saving is 21%, the saving of execution time 6.3%. In this program 12% of execution time is spent in line 49, reading

if ongebruikt[steen] **then**

Here the book–keeping costs are 7.5 μ sec per execution of the line, to be compared with an average value of 146.4 μ sec for the execution of the program code proper, leading to an overhead of 5.1%. The object code of line 49 is 7 instructions with, and 5 instructions without book–keeping. In program length book–keeping leads here to an overhead of 40%!

For ‘lisp interpreter’ the savings are 27% in code length and 6.5% in execution time. Here 14.7% of execution time is spent in line 161, reading

if $x \div d24 = y \div d24$

Again line–number book–keeping costs 7.5 μ sec, whereas the program code proper takes on the average 395.3 μ sec. Hence the time overhead here is 1.9% only, due to the fact that integer division is such an expensive operation, taking the execution of 43 to 59 instructions. In program length the overhead is 2 upon 11, i.e. 18.2%. In line 162, taking 9.2% of the execution time, the time overhead is 7.5 μ sec upon 295.5 μ sec, an overhead of 2.5%. Here again it is the integer division which makes the time saving relatively modest. More typical are perhaps lines 127 and 133, taking together 4.3% of the execution time, where the book–keeping costs of 12.5 μ sec⁸ count heavily in respect of the 90 μ sec⁹ for the action proper. Or line 141, 3.0% of the execution time, with 12.5 μ sec versus 283.75 μ sec.

⁸The code for the last statement of a function restores the line number of the calling side. This costs an additional 5 μ sec.

⁹Including an instruction to deliver the function result in register F.

Smaller savings, on the other hand, are met in ‘Havie integrator’, ‘QR eigenvalues’, and ‘JAZZ164’. These programs have statements with relatively complicated expressions.

For ‘QR eigenvalues’ we measured savings of 16% in program length and 2.3% in execution time. Let us look at the statement in line 67, reading:

$$g[i,j]:= g[i,j] - g[i,k]\times p[j] - p[i]\times g[j,k]$$

responsible for 8.5% of the execution time. Here the program-length overhead is 2 upon 40 instructions, i.e. 5.0%, whereas the execution-time overhead is 7.5 upon 1271.1 μsec , i.e. 0.6%.

For ‘Havie integrator’ with savings of 20% in code length and 0.5% in execution time, 96.2% of the time is spent in line 34, reading:

begin x:= x + h; sumu:= sumu + integrand; count:= count + 1 **end**

where x and integrand are formal parameters (in a construction called ‘Jensen’s device’). For this statement code-length overhead is 4 upon 19 instructions¹⁰, i.e. 21% and execution overhead is 10 upon 2814.2 μsec , i.e. 0.36%.

An extreme case is met in ‘JAZZ164’, where line 172 is responsible for 45.7% of execution time. It reads:

for i:= a **step** 1 **until** b **do** s:= s + xi;

Again i and xi are formal parameters; the actual parameter for xi can be found in line 27 and reads¹¹:

$$m[j]*((y[3\times(j-i)+k]-y[k])\times\text{ownd}[i,j]-y[3*(j-i)+k]\times\text{ownr}[j])$$

The overhead in code length for line 172 was measured to be 2 upon 27 instructions, i.e. 7.4%, that in execution time is 7.5 upon 5841.9 μsec , i.e. 0.13%.

¹⁰At the end of the code for line 34 line number 33 is reinstated, c.f. Section 5.2.7.

¹¹The execution times for this expression contribute to that of line 172.

Chapter 7

Final remarks

In the preceding chapters we described and compared two ALGOL 60 implementations. The first one was developed for the Electrologica X1, a computer designed in the 1950's. The first X1 was delivered in 1958 and its ALGOL 60 implementation was completed in 1960.

The second one was developed for the Electrologica X8, Electrologica's upward compatible successor for the X1. The first X8 was delivered in 1965, as was its ALGOL 60 implementation.

In the 5 years between the completion of the two compilers we see an enormous progress in both hardware and software. Progress in hardware was big. The X8 was, in principle, 12 times as fast as the X1. A memory cycle took 2.5 μ sec compared to 32 μ sec for the X1. Integer division (using registers A and S) took 40 μ sec instead of 500 μ sec.

The ALGOL 60 compiler for the X8, however, compiled programs only about 6 to 8 times as fast but execution speed was increased by a factor 40 to 60.

The compiler speed was strongly influenced by the fact that the X8 version did a thorough syntax check, something almost completely absent in the X1 version. In the 5 years between the completion of the systems the requirements changed drastically and due to the emergence of compiler techniques it was possible to meet these demands. The fact that the compiler for the X8 was designed in ALGOL 60 first and only after completion thereof was recoded in ELAN had undoubtedly a strong influence on its structure.

The execution speed was the result of a number of factors.

1. The X1 object code resembled more or less the instruction code for a hypothetical

stack machine, whereas the X8 object code was in terms of X8 instructions.

2. Results of expressions were delivered in a register rather than on top of stack. Intermediate results were stacked only when the register was to be used for elaboration of other subexpressions.
3. The availability of a floating-point register and hardware floating-point operations in the X8.
4. Hardware support for push and pull operations on a stack.
5. Hardware support for addressing local variables of blocks.
6. Hardware support for the ALGOL 60 parameter mechanism.

Clearly we see here an illustration of the progress in both hardware and software.

Both ALGOL 60 implementations have in common that the job was carried out in about 4 man-year and that the number of flaws was minimal. The compilers were small: 2K and 5K, for the X1 and the X8 respectively. Also the language definition of ALGOL 60 was small: it was defined in a report of 17 pages! Small can be nice and effective, something to keep in mind in this time of mega language definitions and mega compilers.

Appendix A

Sample programs

This Appendix contains the results of running the six programs discussed in Section 6 on an emulator for the X8.

The emulator was loaded with an adapted version of the Pico monitor and an almost unchanged version of the ALGOL 60 system. The original versions were published in [11]. We first discuss here some of the adaptations.

The adapted version of Pico has no interaction with the operator teletype. The date is fictitious: May 2nd, 2005. Each run gets a fixed ‘serial number’: 1. Insertion of line number instructions in object programs (see Section 5.2.6.) is set to be wanted. The only output device is a version of the line printer, adapted to be coded by eight-bit ASCII code rather than the special six-bit line-printer code of the printer. This implies that only three characters can be stored in one X8 word and that the possibility to print lines without paper advancement is abandoned¹.

The most important adaptation of the ALGOL 60 system is in the lower level of the lexical scanner of the compiler. It allows besides the ‘underline style’ for word delimiters also ‘apostrophe style’. This is a later version of the lexical scan which was useful for punch-card equipment: punch cards lend themselves badly to the underline style which led to nice Flexowriter print-outs. Furthermore the output routines for the paper-tape punch are omitted.

The emulator, written in standard Pascal, has a very simple user interface. Pascal text file ‘input’ is used as tape reader and has to contain (Flexowriter) characters coded as numbers (from 0 to 255). Pascal text file ‘output’ is used for line-printer output coded

¹This possibility was used to add underlinings to a line, resulting in ‘begin’ instead of ‘_b_e_g_i_n’.

in ASCII. The emulator contains a version of ‘Charon’, the transport processor of the X8 (cf. Section 1.2.).

Preceding the emulation of each X8 instruction the emulator checks whether there is an activation request for Charon. If so, the requested action is executed completely, including the setting of an interrupt request for the main processor (in the real hardware the requested action was executed in parallel to the activities of the main processor on the basis of cycle stealing). After the care for Charon activities the interrupt requests are inspected and if necessary the corresponding interrupt instruction is selected. Otherwise the next instruction is emulated.

Each instruction is timed and counted. The timing for floating-point instructions is taken from a model (cf. Section 4.6.). Moreover, during execution of object programs the execution time of each instruction is also added to that element of an execution profile array that corresponds to the current line number. Likewise the instruction is counted there. It is here that an emulator creates profiling capabilities that were absent in the real hardware!

The Charon activities in the X8 influenced program execution on the basis of cycle stealing at unpredictable moments. For that reason we ignored this influence in the emulator. Interrupt programs were executed at unpredictable moments too. Their execution times, as well as the number of instructions executed in interrupt programs, are included in the summing up of compilation and execution data, but excluded from the profile.

During compilation an ALGOL 60 program was always listed. If no errors were found it was immediately executed. The execution output started on a new page. In the example programs we find therefore both the program listings and the execution results. After execution completion the emulator added the results of its measurements including the profile.

In general the interpretation of the execution profiles do not present much difficulty. Sometimes, however, some knowledge of the way in which the line numbers are updated might be useful. In the first example, see Appendix A.1., there are no counts for lines 1 upto 24. The reservation of store for simple local variables of the outermost block(s) is already carried out by the compiler. The reservation of simple local variables of a procedure and the evaluation of value parameters are true enough carried out by the code of the procedure, but, since line numbers are set only at the beginning of statements, these activities are counted at the call side (see, e.g., line 72).

In one case some adaptations were necessary to adapt a program written for the X1 ALGOL 60 system to that for the X8. In program ‘JAZZ164’ (Appendix A.3) the local

own arrays 'd' and 'r' of procedure 'f' have been replaced by global arrays 'ownd' and 'ownr', respectively². Moreover, a third parameter is added to calls of the built-in output procedure FLOT for the specification of the number of decimals in the decimal exponent³.

Program 'Lisp interpreter' contains declarations for procedures 'RESYM' and 'PRSYM' in terms of the built-in procedures 'read' and 'PRINTTEXT', respectively. These were added in order to make execution on the X1 possible: the X8 ALGOL 60 system has built-in procedures 'RESYM' and 'PRSYM', but the X1 system doesn't.

²Since in the X1 system own arrays are implemented as global arrays, this adaptation does not affect the comparison of execution times.

³In the X1 system output procedure FLOT had two parameters only, one to specify the number of decimals of the mantissa and one for the number to be printed. The number of decimals for the exponent was here fixed to two.

A.1 Havie integrator

160205 - 1

```

1  _b_e_g_i_n  _c_o_m_m_e_n_t  HAVIE INTEGRATOR.
2              ALGORITHM 257, Robert N. Kubik,
3              CACM 8 (1965) 381;
4
5  _r_e_a_l  a,b,eps,mask,y,answer; _i_n_t_e_g_e_r  count;
6
7  _r_e_a_l  _p_r_o_c_e_d_u_r_e  havieintegrator(x,a,b,eps,integrand,m);
8  _v_a_l_u_e  a,b,eps,m;
9  _i_n_t_e_g_e_r  m; _r_e_a_l  integrand,x,a,b,eps;
10 _c_o_m_m_e_n_t  This algorithm performs numerical integration of
11 definite integrals using an equidistant sampling of the
12 function and repeated halving of the sampling interval.
13 Each halving allows the calculation of a trapezium and
14 a tangent formula on a finer grid, but also the calcul-
15 ation of several higher order formulas which are defined
16 implicitly. The two families of approximate solutions
17 will normally bracket the value of the integral and from
18 these convergence is tested on each of the several orders
19 of approximation. The algorithm is based on a private
20 communication from F. Haavie of the Institutt for Atom-
21 energi Kjeller Research Establishment, Norway. A Fortran
22 version is in use on the Philco-2000. ...;
23 _b_e_g_i_n  _r_e_a_l  h,endpts,sumt,sumu,d;
24 _i_n_t_e_g_e_r  i,j,k,n;
25 _r_e_a_l  _a_r_r_a_y  t,u,tprev,uprev[1:m];
26 x:= a; endpts:= integrand;          count:= 1;
27 x:= b; endpts:= 0.5 * (integrand + endpts);  count:= count + 1;
28 sumt:= 0.0; i:= n:= 1; h:= b - a;
29 estimate:
30 t[1]:= h * (endpts + sumt); sumu:= 0.0;
31 _c_o_m_m_e_n_t  t[1] = h*(0.5*f[0]+f[1]+f[2]+...+0.5*f[2^(i-1)]);
32 x:= a - h/2.0;
33 _f_o_r  j:= 1 _s_t_e_p 1 _u_n_t_i_l  n _d_o
34 _b_e_g_i_n  x:= x + h; sumu:= sumu + integrand; count:= count + 1 _e_n_d;
35 u[1]:= h * sumu; k:= 1;
36 _c_o_m_m_e_n_t  u[1] = h*(f[1/2]+f[3/2]+...f[(2^i-1)/2],
37 k corresponds to approximate solution with truncation
38 error term of order 2k;
39 test:
40 _i_f  abs(t[k]-u[k]) < eps
41 _t_h_e_n  _b_e_g_i_n  havieintegrator:= 0.5 * (t[k] + u[k]);
42 _g_o_t_o  exit
43 _e_n_d;
44 _i_f  k |= i
45 _t_h_e_n  _b_e_g_i_n  d:= 2 |^(2*k);
46 t[k+1]:= (d * t[k] - tprev[k]) / (d - 1.0);
47 tprev[k]:= t[k];
48 u[k+1]:= (d * u[k] - uprev[k]) / (d - 1.0);
49 uprev[k]:= u[k];
50 _c_o_m_m_e_n_t  This implicit formulation of the higher
51 order integration formulas is given in
52 [ROMBERG, W. ...;
53 k:= k + 1;
54 _i_f  k = m
55 _t_h_e_n  _b_e_g_i_n  havieintegrator:= mask;
56 _g_o_t_o  exit
57 _e_n_d;
58 _g_o_t_o  test
59 _e_n_d;

```

```
160205 -      1

60      h:= h / 2.0; sumt:= sumt + sumu;
61      tprev[k]:= t[k]; uprev[k]:= u[k];
62      i:= i + 1; n:= 2 * n;
63      _g_o_t_o estimate;
64      exit: NLCR; NLCR;
65      PRINTTEXT(|<i: |>); ABSFIXT(3,0,i);
66      PRINTTEXT(|<k: |>); ABSFIXT(3,0,k);
67      PRINTTEXT(|< count: |>); ABSFIXT(7,0,count)
68      _e_n_d havieintegrator;
69
70      _c_o_m_m_e_n_t Following is a driver program to test havieintegrator;
71      a:= 0.0; b:= 1.5707963; eps:= 0.00001; mask:= 9.99;
72      answer:= havieintegrator(y,a,b,eps,cos(y),12);
73      NLCR; PRINTTEXT(|<integral(0,pi/2,cos(x)) = |>);
74      FLOT(10,2,answer);
75      a:= 0.0; b:= 4.3;
76      answer:= havieintegrator(y,a,b,eps,exp(-y*y),12);
77      NLCR; PRINTTEXT(|<integral(0,4.3,exp(-x*x)) = |>);
78      FLOT(10,2,answer);
79      a:= 1.0; b:= 10.0;
80      answer:= havieintegrator(y,a,b,eps,ln(y),12);
81      NLCR; PRINTTEXT(|<integral(1,10,ln(x)) = |>);
82      FLOT(10,2,answer);
83      a:= 0.0; b:= 20.0;
84      answer:= havieintegrator(y,a,b,eps,y^(1/2)/(exp(y-4)+1),20);
85      NLCR; PRINTTEXT(|<integral(1,20,x^(1/2)/(exp(x-4)+1)) = |>);
86      FLOT(10,2,answer);
87      _e_n_d
88
```

160205 - 1

```
i: 4 k: 3 count: 17
integral(0,pi/2,cos(x)) = +.999999981%- 0

i: 4 k: 1 count: 17
integral(0,4.3,exp(-x*x)) = +.8862269239%- 0

i: 7 k: 2 count: 129
integral(1,10,ln(x)) = +.1402585066%+ 2

i: 15 k: 2 count: 32769
integral(1,20,x^(1/2)/(exp(x-4)+1)) = +.5770724810%+ 1
```

160205 - 1

compiler data

number of instructions executed: 787624
compile time (microseconds): 3084811.25
average instruction time: 3.92
object--program length: 609

execution data

number of instructions executed: 9788838
execution time (microseconds): 96701315.00
average instruction time: 9.88

profile

linenumber	count	time	%
25	724	3310.00	0.0
26	329	2700.00	0.0
27	546	5595.00	0.0
28	48	317.50	0.0
30	990	6471.25	0.0
32	300	4243.75	0.0
33	560218	3335992.50	3.4
34	9158426	92983682.50	96.2
35	960	5625.00	0.0
40	6132	31537.50	0.0
41	160	855.00	0.0
42	12	40.00	0.0
44	852	4082.50	0.0
45	6812	34945.00	0.0
46	8236	53240.00	0.1
47	4872	24070.00	0.0
48	8236	53240.00	0.1
49	4872	24070.00	0.0
53	812	3770.00	0.0
54	580	2900.00	0.0
58	348	1160.00	0.0
60	208	3233.75	0.0
61	2132	10595.00	0.0
62	312	1730.00	0.0
63	78	260.00	0.0
64	296	1280.00	0.0
65	2534	12001.25	0.0
66	2532	11941.25	0.0
67	3872	18828.75	0.0
71	10	67.50	0.0
72	203	830.00	0.0
73	1112	4746.25	0.0
74	979	5616.25	0.0
75	6	37.50	0.0
76	203	830.00	0.0
77	1112	4746.25	0.0
78	979	5595.00	0.0
79	6	37.50	0.0
80	203	830.00	0.0
81	1112	4746.25	0.0
82	1005	5895.00	0.0
83	6	37.50	0.0
84	203	830.00	0.0
85	1476	6308.75	0.0
86	1005	5910.00	0.0
87	102	463.75	0.0

A.2 QR eigenvalues

160205 - 1

```

1  _b_e_g_i_n _c_o_m_m_e_n_t eigenvalues of a real symmetric matrix
2  by the QR method. Algorithm 253, P.A. Businger, CACM 8 (1965) 217;
3
4  _i_n_t_e_g_e_r n;
5
6  n:= read;
7  _b_e_g_i_n _i_n_t_e_g_e_r i,j;
8  _r_e_a_l _a_r_r_a_y a[1:n,1:n];
9
10 _p_r_o_c_e_d_u_r_e symmetric QR1(n,g);
11 _v_a_l_u_e n; _i_n_t_e_g_e_r n; _a_r_r_a_y g;
12
13 _c_o_m_m_e_n_t uses Housholders's method and the QR algorithm to
14 find all n eigenvalues of the real symmetric matrix whose lower
15 triangular part is given in the array g[1:n,1:n]. The computed
16 eigenvalues are stored as the diagonal elements g[i,i]. The
17 original contents of the lower triangular part of g are lost during
18 the computation whereas the strictly upper triangular part of g
19 is left untouched;
20
21 _b_e_g_i_n
22
23 _r_e_a_l _p_r_o_c_e_d_u_r_e sum(i,m,n,a);
24 _v_a_l_u_e m,n; _i_n_t_e_g_e_r i,m,n; _r_e_a_l a;
25 _b_e_g_i_n _r_e_a_l s; s:= 0;
26 _f_o_r i:= m _s_t_e_p 1 _u_n_t_i_l n _d_o s:= s + a;
27 sum:= s
28 _e_n_d sum;
29
30 _r_e_a_l _p_r_o_c_e_d_u_r_e max (a,b);
31 _v_a_l_u_e a,b; _r_e_a_l a,b;
32 max:= _i_f a > b _t_h_e_n a _e_l_s_e b;
33
34 _p_r_o_c_e_d_u_r_e Housholder tridiagonalization 1(n,g,a,bq,norm);
35 _v_a_l_u_e n; _i_n_t_e_g_e_r n; _a_r_r_a_y g,a,bq; _r_e_a_l norm;
36 _c_o_m_m_e_n_t nonlocal real procedure sum, max;
37 _c_o_m_m_e_n_t reduces the given real symmetric n by n matrix g
38 to tridiagonal form using n - 2 elementary orthogonal trans-
39 formations (I-2ww') = (I-gamma uu'). Only the lower tri-
40 angular part of g need be given. The diagonal elements and
41 the squares of the subdiagonal elements of the reduced matrix
42 are stored in a[1:n] and bq[1:n-1] respectively. norm is set
43 equal to the infinity norm of the reduced matrix. The columns
44 of the strictly lower triangular part of g are replaced by the
45 nonzero portions of the vectors u;
46 _b_e_g_i_n _i_n_t_e_g_e_r i,j,k;
47 _r_e_a_l t,absb,alpha,beta,gamma,sigma;
48 _a_r_r_a_y p[2:n];
49 norm:= absb:= 0;
50 _f_o_r k:= 1 _s_t_e_p 1 _u_n_t_i_l n - 2 _d_o
51 _b_e_g_i_n a[k]:= g[k,k];
52 sigma:= bq[k]:= sum(i,k+1,n,g[i,k]^2);
53 t:= absb + abs(a[k]); absb:= sqrt(sigma);
54 norm:= max(norm,t+absb);
55 _i_f sigma != 0 _t_h_e_n
56 _b_e_g_i_n alpha:= g[k+1,k];
57 beta:= _i_f alpha < 0 _t_h_e_n absb _e_l_s_e - absb;
58 gamma:= 1 / (sigma-alpha*beta); g[k+1,k]:= alpha - beta;
59 _f_o_r i:= k + 1 _s_t_e_p 1 _u_n_t_i_l n _d_o

```

```

160205 -      1

60      p[i]:= gamma *
61          (sum(j,k+1,i,g[i,j]*g[j,k]) + sum(j,i+1,n,g[j,i]*g[j,k]));
62      t:= 0.5 * gamma * sum(i,k+1,n,g[i,k]*p[i]);
63      _f_o_r i:= k + 1 _s_t_e_p 1 _u_n_t_i_l n _d_o
64      p[i]:= p[i] - t*g[i,k];
65      _f_o_r i:= k + 1 _s_t_e_p 1 _u_n_t_i_l n _d_o
66      _f_o_r j:= k + 1 _s_t_e_p 1 _u_n_t_i_l i _d_o
67      g[i,j]:= g[i,j] - g[i,k]*p[j] - p[i]*g[j,k]
68
69      _e_n_d k;
70      a[n-1]:= g[n-1,n-1]; bq[n-1]:= g[n,n-1]^2;
71      a[n]:= g[n,n]; t:= abs(g[n,n-1]);
72      norm:= max(norm,absb+abs(a[n-1])+t);
73      norm:= max(norm,t+abs(a[n]))
74      _e_n_d Housholder tridiagonalization 1;
75
76      _i_n_t_e_g_e_r i,k,m,m1;
77      _r_e_a_l norm,epsq,lambda,mu,sq1,sq2,u,pq,gamma,t;
78      _a_r_r_a_y a[1:n],bq[0:n-1];
79
80      Housholder tridiagonalization 1(n,g,a,bq,norm);
81      epsq:= 2.25%-22*norm^2;
82      _c_o_m_m_e_n_t The tolerance used in the QR iteration depends
83      on the square of the relative machine precision. Here 2.25%-22
84      is used which is appropriate for a machine with a 36-bit
85      mantissa;
86      mu:= 0; m:= n;
87      inspect: _i_f m = 0
88      _t_h_e_n _g_o_t_o return _e_l_s_e i:= k:= m1:= m - 1;
89      bq[0]:= 0;
90      _i_f bq[k] < epsq _t_h_e_n
91      _b_e_g_i_n g[m,m]:= a[m]; mu:= 0; m:= k;
92      _g_o_t_o inspect
93      _e_n_d;
94      _f_o_r i:= i - 1 _w_h_i_l_e bq[i] > epsq _d_o k:= i;
95      _i_f k = m1 _t_h_e_n
96      _b_e_g_i_n _c_o_m_m_e_n_t treat 2 * 2 block separately;
97      mu:= a[m1]*a[m] - bq[m1]; sq1:= a[m1] + a[m];
98      sq2:= sqrt((a[m1]-a[m])^2+4*bq[m1]);
99      lambda:= 0.5*(i_f sq1>0 _t_h_e_n sq1+sq2 _e_l_s_e sq1-sq2);
100     g[m1,m1]:= lambda; g[m,m]:= mu / lambda;
101     mu:= 0; m:= m - 2; _g_o_t_o inspect
102     _e_n_d;
103     lambda:= _i_f abs(a[m]-mu) < 0.5*abs(a[m])
104     _t_h_e_n a[m] + 0.5*sqrt(bq[m1]) _e_l_s_e 0.0;
105     mu:= a[m]; sq1:= sq2:= u:= 0;
106     _f_o_r i:= k _s_t_e_p 1 _u_n_t_i_l m1 _d_o
107     _b_e_g_i_n _c_o_m_m_e_n_t shortcut single QR iteration;
108     gamma:= a[i] - lambda - u;
109     pq:= _i_f sq1 | = 1
110     _t_h_e_n gamma|^2/(1-sq1) _e_l_s_e (1-sq2)*bq[i-1];
111     t:= pq + bq[i]; bq[i-1]:= sq1 * t; sq2:= sq1;
112     sq1:= bq[i] / t; u:= sq1 * (gamma+a[i+1]-lambda);
113     a[i]:= gamma + u + lambda
114     _e_n_d i;
115     gamma:= a[m] - lambda - u;
116     bq[m1]:= sq1 *
117     (i_f sq1|=1 _t_h_e_n gamma|^2/(1-sq1) _e_l_s_e (1-sq2)*bq[m1]);
118     a[m]:= gamma + lambda; _g_o_t_o inspect;
119     return;

```

```
160205 -      1
120      _e_n_d symmetric QR 1;
121
122
123      _f_o_r i:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o
124      _f_o_r j:= 1 _s_t_e_p 1 _u_n_t_i_l i _d_o
125          a[i,j]:= read;
126
127      symmetric QR 1(n,a);
128
129      _f_o_r i:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o
130      _b_e_g_i_n NLCR; ABSFIXT(2,0,i); PRINT(a[i,i]) _e_n_d
131
132      _e_n_d
133
134      _e_n_d
135
```

160205 - 1

1 +.2240687530767%+ 2
2 +.7513724154203%+ 1
3 +.4848950120329%+ 1
4 -.1096595181629%+ 1
5 +.1327045599557%+ 1

160205 - 1

compiler data

number of instructions executed: 1240976
 compile time (microseconds): 4865792.50
 average instruction time: 3.92
 object--program length: 1211

execution data

number of instructions executed: 51363
 execution time (microseconds): 284983.75
 average instruction time: 5.55

profile

linenumber	count	time	%
6	288	1137.50	0.4
7	12	48.75	0.0
8	94	443.75	0.2
25	96	480.00	0.2
26	6466	36425.00	12.8
27	120	1020.00	0.4
32	48	365.00	0.1
48	67	306.25	0.1
49	6	45.00	0.0
50	72	400.00	0.1
51	264	1280.00	0.4
52	660	2816.25	1.0
53	255	1687.50	0.6
54	351	1607.50	0.6
55	18	87.50	0.0
56	156	758.75	0.3
57	28	162.50	0.1
58	219	1523.75	0.5
59	207	1181.25	0.4
60	3501	15241.25	5.3
62	558	2543.75	0.9
63	207	1181.25	0.4
64	864	4766.25	1.7
65	207	1181.25	0.4
66	485	2733.75	1.0
67	4636	24293.75	8.5
70	221	1095.00	0.4
71	140	682.50	0.2
72	152	708.75	0.2
73	155	716.25	0.3
78	132	598.75	0.2
80	177	722.50	0.3
81	49	341.25	0.1
86	8	41.25	0.0
87	221	1123.75	0.4
88	3	10.00	0.0
89	486	2340.00	0.8
90	378	1863.75	0.7
91	240	1170.00	0.4
92	9	30.00	0.0
94	1305	6662.50	2.3
95	75	375.00	0.1
96	2	7.50	0.0
97	91	512.50	0.2
98	146	885.00	0.3
99	13	93.75	0.0
100	117	645.00	0.2
101	10	47.50	0.0
103	1492	9022.50	3.2
105	322	1750.00	0.6
106	986	5662.50	2.0
107	88	330.00	0.1
108	924	5176.25	1.8
109	2508	17138.75	6.0
111	2244	14517.50	5.1
112	1980	16133.75	5.7
113	1276	7405.00	2.6
115	294	1660.00	0.6
116	1162	8236.25	2.9

118	406	2212.50	0.8
120	7	30.00	0.0
123	102	530.00	0.2
124	340	1737.50	0.6
125	3225	14217.50	5.0
127	83	326.25	0.1
129	203	993.75	0.3
130	8086	46241.25	16.2

A.3 JAZZ164, Runge–Kutta integration

160205 - 1

```

1  _b_e_g_i_n _c_o_m_m_e_n_t JAZZ164, R743, Outer Planets;
2
3  _i_n_t_e_g_e_r k,t; _r_e_a_l a,k2,x; _B_o_o_l_e_a_n fi;
4  _a_r_r_a_y y,ya,z,za[1:15],m[0:5],e[1:60],d[1:33];
5
6  _a_r_r_a_y ownd[1:5,1:5],ownr[1:5];
7
8  _r_e_a_l _p_r_o_c_e_d_u_r_e f(k); _i_n_t_e_g_e_r k;
9  _b_e_g_i_n _i_n_t_e_g_e_r i,j,i3,j3; _r_e_a_l p;
10 _i_f k != 1 _t_h_e_n _g_o_t_o A;
11 _f_o_r i:= 1 _s_t_e_p 1 _u_n_t_i_l 4 _d_o
12   _b_e_g_i_n i3:= 3*i;
13   _f_o_r j:= i+1 _s_t_e_p 1 _u_n_t_i_l 5 _d_o
14     _b_e_g_i_n j3:= 3*j;
15     p:= (y[i3-2] - y[j3-2])|^2 + (y[i3-1] - y[j3-1])|^2 +
16         (y[i3] - y[j3])|^2;
17     ownd[i,j]:= ownd[j,i]:= 1/p/sqrt(p)
18   _e_n_d
19 _e_n_d ;
20 _f_o_r i:= 1 _s_t_e_p 1 _u_n_t_i_l 5 _d_o
21   _b_e_g_i_n i3:= 3*i; ownd[i,i]:= 0;
22   p:= y[i3-2]^2 + y[i3-1]^2 + y[i3]^2;
23   ownr[i]:= 1/p/sqrt(p)
24 _e_n_d ;
25 A: i:= (k - 1) _: 3 + 1;
26   f:= k2 * (- m[0] * y[k] * ownr[i] +
27   SUM(j,1,5,m[j]*((y[3*(j-i)+k]-y[k])*ownd[i,j]-y[3*(j-i)+k]*ownr[j])))
28 _e_n_d f;
29
30 _p_r_o_c_e_d_u_r_e RK3n(x,a,b,y,ya,z,za,fxj,j,e,d,fi,n);
31 _v_a_l_u_e b,fi,n; _i_n_t_e_g_e_r j,n; _r_e_a_l x,a,b,fxj;
32 _B_o_o_l_e_a_n fi; _a_r_r_a_y y,ya,z,za,e,d;
33 _b_e_g_i_n _i_n_t_e_g_e_r jj;
34 _r_e_a_l xl,h,hmin,int,hl,absh,fhm,discry,discrz,toly,tolz,mu,mu1,fhy,fhz;
35 _B_o_o_l_e_a_n last,first,reject;
36 _a_r_r_a_y yl,zl,k0,k1,k2,k3,k4,k5[1:n],ee[1:4*n];
37 _i_f fi
38   _t_h_e_n _b_e_g_i_n d[3]:= a;
39     _f_o_r jj:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o
40       _b_e_g_i_n d[jj+3]:= ya[jj]; d[n+jj+3]:= za[jj] _e_n_d
41     _e_n_d ;
42     d[1]:= 0; xl:= d[3];
43     _f_o_r jj:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o
44       _b_e_g_i_n yl[jj]:= d[jj+3]; zl[jj]:= d[n+jj+3] _e_n_d ;
45     _i_f fi _t_h_e_n d[2]:= b - d[3];
46     absh:= h:= abs(d[2]);
47     _i_f b - xl < 0 _t_h_e_n h:= - h;
48     int:= abs(b - xl); hmin:= int * e[1] + e[2];
49     _f_o_r jj:= 2 _s_t_e_p 1 _u_n_t_i_l 2*n _d_o
50       _b_e_g_i_n hl:= int * e[2*jj-1] + e[2*jj];
51       _i_f hl < hmin _t_h_e_n hmin:= hl
52     _e_n_d ;
53     _f_o_r jj:= 1 _s_t_e_p 1 _u_n_t_i_l 4*n _d_o ee[jj]:= e[jj]/int;
54     first:= reject:= _t_r_u_e ;
55     _i_f fi
56       _t_h_e_n _b_e_g_i_n last:= _t_r_u_e ; _g_o_t_o step _e_n_d ;
57 test: absh:= abs(h);
58 _i_f absh < hmin
59   _t_h_e_n _b_e_g_i_n h:= _i_f h > 0 _t_h_e_n hmin _e_l_s_e - hmin;

```

```

160205 -      1

60      absh:= hmin
61      _e_n_d ;
62      _i_f h > b - xl _= h > 0
63      _t_h_e_n _b_e_g_i_n d[2]:= h; last:= _t_r_u_e ;
64      h:= b - xl; absh:= abs(h)
65      _e_n_d
66      _e_l_s_e last:= _f_a_l_s_e ;
67      step: _i_f reject
68      _t_h_e_n _b_e_g_i_n x:= xl;
69      _f_o_r jj:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o
70      y[jj]:= y1[jj];
71      _f_o_r j:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o
72      k0[j]:= fxyj * h
73      _e_n_d
74      _e_l_s_e _b_e_g_i_n fhy:= h/hl;
75      _f_o_r jj:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o
76      k0[jj]:= k5[jj] * fhy
77      _e_n_d ;
78      x:= xl + .27639 32022 50021 * h;
79      _f_o_r jj:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o
80      y[jj]:= y1[jj] + (z1[jj] * .27639 32022 50021 +
81      k0[jj] * .03819 66011 25011) * h;
82      _f_o_r j:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o k1[j]:= fxyj * h;
83      x:= xl + .72360 67977 49979 * h;
84      _f_o_r jj:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o
85      y[jj]:= y1[jj] + (z1[jj] * .72360 67977 49979 +
86      k1[jj] * .26180 33988 74989) * h;
87      _f_o_r j:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o k2[j]:= fxyj * h;
88      x:= xl + h * .5;
89      _f_o_r jj:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o
90      y[jj]:= y1[jj] + (z1[jj] * .5 +
91      k0[jj] * .04687 5 +
92      k1[jj] * .07982 41558 39840 -
93      k2[jj] * .00169 91558 39840) * h;
94      _f_o_r j:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o k4[j]:= fxyj * h;
95      x:= _i_f last _t_h_e_n b _e_l_s_e xl + h;
96      _f_o_r jj:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o
97      y[jj]:= y1[jj] + (z1[jj] +
98      k0[jj] * .30901 69943 74947 +
99      k2[jj] * .19098 30056 25053) * h;
100     _f_o_r j:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o k3[j]:= fxyj * h;
101     _f_o_r jj:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o
102     y[jj]:= y1[jj] + (z1[jj] +
103     k0[jj] * .08333 33333 33333 +
104     k1[jj] * .30150 28323 95825 +
105     k2[jj] * .11516 38342 70842) * h;
106     _f_o_r j:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o k5[j]:= fxyj * h;
107     reject:= _f_a_l_s_e ; fhm:= 0;
108     _f_o_r jj:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o
109     _b_e_g_i_n
110     discry:= abs((- k0[jj] * .5 + k1[jj] * 1.80901 69943 74947 +
111     k2[jj] * .69098 30056 25053 - k4[jj] * 2) * h);
112     discrz:= abs((k0[jj] - k3[jj]) * 2 - (k1[jj] + k2[jj]) * 10 +
113     k4[jj] * 16 + k5[jj] * 4);
114     toly:= absh * (abs(z1[jj]) * ee[2*jj-1] + ee[2*jj]);
115     tolz:= abs(k0[jj]) * ee[2*(jj+n)-1] + absh * ee[2*(jj+n)];
116     reject:= discry > toly # discrz > tolz # reject;
117     fhy:= discry/toly; fhz:= discrz/to lz;
118     _i_f fhz > fhy _t_h_e_n fhy:= fhz;
119     _i_f fhy > fhm _t_h_e_n fhm:= fhy

```

```

160205 - 1

120  _e_n_d ;
121  mu:= 1/(1 + fhm) + .45;
122  _i_f reject
123  _t_h_e_n _b_e_g_i_n _i_f absh < hmin
124      _t_h_e_n _b_e_g_i_n d[1]:= d[1] + 1;
125          _f_o_r jj:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o
126              _b_e_g_i_n y[jj]:= yl[jj];
127                  z[jj]:= zl[jj]
128                      _e_n_d ;
129                          first:= _t_r_u_e ; _g_o_t_o next
130                              _e_n_d ;
131                                  h:= mu * h; _g_o_t_o test
132                                      _e_n_d rej;
133  _i_f first
134  _t_h_e_n _b_e_g_i_n first:= _f_a_l_s_e ; hl:= h; h:= mu * h;
135      _g_o_t_o acc
136      _e_n_d ;
137  fhy:= mu * h/hl + mu - mu1; hl:= h; h:= fhy * h;
138  acc: mu1:= mu;
139  _f_o_r jj:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o
140  z[jj]:= zl[jj] + (k0[jj] + k3[jj]) * .08333 33333 33333 +
141      (k1[jj] + k2[jj]) * .41666 66666 66667;
142  next: _i_f b | = x
143      _t_h_e_n _b_e_g_i_n xl:= x;
144          _f_o_r jj:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o
145              _b_e_g_i_n yl[jj]:= y[jj]; zl[jj]:= z[jj] _e_n_d ;
146                  _g_o_t_o test
147                      _e_n_d ;
148  _i_f ~ last _t_h_e_n d[2]:= h;
149  d[3]:= x;
150  _f_o_r jj:= 1 _s_t_e_p 1 _u_n_t_i_l n _d_o
151  _b_e_g_i_n d[jj+3]:= y[jj]; d[n+jj+3]:= z[jj] _e_n_d
152  _e_n_d RK3n;
153
154  _p_r_o_c_e_d_u_r_e TYP(x); _a_r_r_a_y x;
155  _b_e_g_i_n _i_n_t_e_g_e_r k;
156  NLCR; PRINTTEXT(|<T = |>); ABSFIXT(7,1,t+a); NLCR; NLCR;
157  _f_o_r k:= 1 _s_t_e_p 1 _u_n_t_i_l 5 _d_o
158  _b_e_g_i_n _i_f k=1 _t_h_e_n PRINTTEXT(|<J |>) _e_l_s_e
159  _i_f k=2 _t_h_e_n PRINTTEXT(|<S |>) _e_l_s_e
160  _i_f k=3 _t_h_e_n PRINTTEXT(|<U |>) _e_l_s_e
161  _i_f k=4 _t_h_e_n PRINTTEXT(|<N |>) _e_l_s_e
162      PRINTTEXT(|<P |>);
163      FIXT(2,9,x[3*k-2]); FIXT(2,9,x[3*k-1]); FIXT(2,9,x[3*k]);
164  NLCR
165  _e_n_d
166  _e_n_d TYP;
167
168  _r_e_a_l _p_r_o_c_e_d_u_r_e SUM(i,a,b,xi);
169  _v_a_l_u_e b; _i_n_t_e_g_e_r i,a,b; _r_e_a_l xi;
170  _b_e_g_i_n _r_e_a_l s;
171  s:= 0;
172  _f_o_r i:= a _s_t_e_p 1 _u_n_t_i_l b _d_o s:= s + xi;
173  SUM:= s
174  _e_n_d SUM;
175
176  a:= read;
177  _f_o_r k:= 1 _s_t_e_p 1 _u_n_t_i_l 15 _d_o
178  _b_e_g_i_n ya[k]:= read; za[k]:= read _e_n_d ;
179  _f_o_r k:= 0 _s_t_e_p 1 _u_n_t_i_l 5 _d_o m[k]:= read;

```

```
160205 -      1
180      k2:= read; e[1]:= read;
181      _f_o_r k:= 2 _s_t_e_p 1 _u_n_t_i_l 60 _d_o e[k]:= e[1];
182      NLCR; PRINTTEXT(|<JAZ164, R743, Outer Planets|>); NLCR; NLCR;
183      _f_o_r k:= 1 _s_t_e_p 1 _u_n_t_i_l 15 _d_o
184      _b_e_g_i_n FLOT(12,2,ya[k]); FLOT(12,2,za[k]); NLCR _e_n_d ;
185      _f_o_r k:= 0 _s_t_e_p 1 _u_n_t_i_l 5 _d_o
186      _b_e_g_i_n NLCR; FLOT(12,2,m[k]) _e_n_d ;
187      NLCR; NLCR; FLOT(12,2,k2);
188      NLCR; NLCR; PRINTTEXT(|<eps = |>); FLOT(2,2,e[1]); NLCR;
189      t:= 0; TYP(ya); fi:= _t_r_u_e ;
190      _f_o_r t:= 500,1000 _d_o
191      _b_e_g_i_n RK3n(x,0,t,y,ya,z,za,f(k),k,e,d,fi,15);
192      fi:= _f_a_l_s_e ; TYP(y)
193      _e_n_d
194      _e_n_d
195
```

160205 - 1

JAZ164, R743, Outer Planets

+ .342947415189%+ 1 - .557160570446%- 2
 + .335386959711%+ 1 + .505696783289%- 2
 + .135494901715%+ 1 + .230578543901%- 2
 + .664145542550%+ 1 - .415570776342%- 2
 + .597156957878%+ 1 + .365682722812%- 2
 + .218231499728%+ 1 + .169143213293%- 2
 + .112630437207%+ 2 - .325325669158%- 2
 + .146952576794%+ 2 + .189706021964%- 2
 + .627960525067%+ 1 + .877265322780%- 3
 - .301552268759%+ 2 - .240476254170%- 3
 + .165699966404%+ 1 - .287659532608%- 2
 + .143785752721%+ 1 - .117219543175%- 2
 - .211238353380%+ 2 - .176860753121%- 2
 + .284465098142%+ 2 - .216393453025%- 2
 + .153882659679%+ 2 - .148647893090%- 3

+ .10000597682%+ 1
 + .954786104043%- 3
 + .285583733151%- 3
 + .437273164546%- 4
 + .517759138449%- 4
 + .277777777778%- 5

+ .295912208286%- 3

eps = + .10%- 3

T = 2430000.5

J	+3.429474152	+3.353869597	+1.354949017
S	+6.641455426	+5.971569579	+2.182314997
U	+11.263043721	+14.695257679	+6.279605251
N	-30.155226876	+1.656999664	+1.437857527
P	-21.123835338	+28.446509814	+15.388265968

T = 2430500.5

J	-.049534455	+4.714982495	+2.023963513
S	+4.277614611	+7.483210480	+2.909418313
U	+9.582290073	+15.567813885	+6.685732380
N	-30.235783049	+2.15924799	+8.49602274
P	-21.994991444	+27.345130515	+15.303485551

T = 2431000.5

J	-3.535429691	+3.610053139	+1.635176964
S	+1.496149963	+8.261862331	+3.351487277
U	+7.805112554	+16.281370897	+7.023579152
N	-30.235569469	-1.228279723	+2.57987477
P	-22.837219187	+26.205087209	+15.197406000

160205 - 1

compiler data

number of instructions executed: 2270261
 compile time (microseconds): 8918110.00
 average instruction time: 3.93
 object--program length: 2497

execution data

number of instructions executed: 1705131
 execution time (microseconds): 10363829.00
 average instruction time: 6.08

profile

linenumber	count	time	%
4	372	1653.75	0.0
6	158	720.00	0.0
10	5616	26190.00	0.3
11	4590	24907.50	0.2
12	1512	8707.50	0.1
13	13068	70335.00	0.7
14	3780	22545.00	0.2
15	130680	690161.25	6.7
17	74520	502625.00	4.8
20	5508	30105.00	0.3
21	13500	68242.50	0.7
22	49410	263942.50	2.5
23	21060	171271.25	1.7
25	25164	170235.00	1.6
26	164430	849295.00	8.2
36	796	3652.50	0.0
37	8	35.00	0.0
38	41	200.00	0.0
39	272	1600.00	0.0
40	2115	10275.00	0.1
42	144	682.50	0.0
43	544	3200.00	0.0
44	3390	16800.00	0.2
45	80	388.75	0.0
46	72	352.50	0.0
47	16	80.00	0.0
48	146	790.00	0.0
49	1080	6860.00	0.1
50	4118	22972.50	0.2
51	406	2465.00	0.0
53	8556	53442.50	0.5
54	14	65.00	0.0
55	8	35.00	0.0
56	6	23.75	0.0
57	54	337.50	0.0
58	63	355.00	0.0
62	144	893.75	0.0
63	88	427.50	0.0
64	20	147.50	0.0
66	35	148.75	0.0
67	40	175.00	0.0
68	20	170.00	0.0
69	1088	6400.00	0.1
70	3360	16200.00	0.2
71	1412	9430.00	0.1
72	5280	27563.75	0.3
74	30	557.50	0.0
75	1632	9600.00	0.1
76	3870	24417.50	0.2
78	90	1227.50	0.0
79	2720	16000.00	0.2
80	14250	94266.25	0.9
82	16120	91021.25	0.9
83	90	1240.00	0.0
84	2720	16000.00	0.2
85	14250	95001.25	0.9
87	16120	91013.75	0.9
88	90	837.50	0.0
89	2720	16000.00	0.2

90	20100	129883.75	1.3
94	16120	91047.50	0.9
95	80	621.25	0.0
96	2720	16000.00	0.2
97	16950	110526.25	1.1
100	16120	91042.50	0.9
101	2720	16000.00	0.2
102	19800	132376.25	1.3
106	16120	91041.25	0.9
107	70	337.50	0.0
108	2720	16000.00	0.2
109	300	1125.00	0.0
110	12300	86973.75	0.8
112	17550	96007.50	0.9
114	9450	67182.50	0.6
115	10350	73322.50	0.7
116	2850	16346.25	0.2
117	1200	26968.75	0.3
118	900	5330.00	0.1
119	928	5652.50	0.1
121	110	1360.00	0.0
122	40	175.00	0.0
123	12	73.75	0.0
131	12	152.50	0.0
133	32	140.00	0.0
134	20	250.00	0.0
135	6	20.00	0.0
137	78	1702.50	0.0
138	32	220.00	0.0
139	2176	12800.00	0.1
140	15600	92141.25	0.9
142	64	452.50	0.0
143	24	195.00	0.0
144	1632	9600.00	0.1
145	9900	49050.00	0.5
146	18	60.00	0.0
148	10	42.50	0.0
149	82	410.00	0.0
150	554	3245.00	0.0
151	4230	20550.00	0.2
156	3201	17405.00	0.2
157	321	1740.00	0.0
158	660	2748.75	0.0
159	645	2685.00	0.0
160	630	2621.25	0.0
161	615	2557.50	0.0
162	588	2445.00	0.0
163	42121	251957.50	2.4
164	570	2456.25	0.0
171	3240	16200.00	0.2
172	730620	4738025.00	45.7
173	4050	34425.00	0.3
176	651	2790.00	0.0
177	272	1402.50	0.0
178	25370	115621.25	1.1
179	5319	24255.00	0.2
180	1415	6156.25	0.1
181	3262	15927.50	0.2
182	1150	4910.00	0.0
183	272	1402.50	0.0
184	34111	202182.50	2.0
185	119	603.75	0.0
186	6915	40701.25	0.4
187	1154	6766.25	0.1
188	1049	4878.75	0.0
189	51	200.00	0.0
190	123	538.75	0.0
191	700	2700.00	0.0
192	94	372.50	0.0

A.4 Sieve of Erathostenes

```

160205 -      1

1  _b_e_g_i_n  _c_o_m_m_e_n_t  zeef van Erathostenes;
2  _i_n_t_e_g_e_r  p, m;
3  _b_o_o_l_e_a_n  _a_r_r_a_y  prime [2 : 10000];
4
5  _f_o_r  p:= 2, 3 _s_t_e_p 2 _u_n_t_i_l 10000 _d_o
6      prime[p]:= _t_r_u_e ;
7
8  _f_o_r  p:= 2, 3 _s_t_e_p 2 _u_n_t_i_l 10 _d_o
9      _b_e_g_i_n  _i_f  prime[p]
10         _t_h_e_n  _f_o_r  m:= p * p _s_t_e_p  p _u_n_t_i_l 10000 _d_o
11             prime[m]:= _f_a_l_s_e
12         _e_n_d;
13
14  m:= 0;
15  _f_o_r  p:= 2, 3 _s_t_e_p 2 _u_n_t_i_l 10000 _d_o
16      _b_e_g_i_n  _i_f  prime[p]
17         _t_h_e_n  _b_e_g_i_n  m:= m + 1 _e_n_d
18         _e_n_d;
19
20  NLCR; PRINTTEXT(|<number of primes below 100 000: |>); ABSFIXT(5,0,m)
21
22  _e_n_d
23

```

```
160205 -      1
```

```
number of primes below 100 000: 2288
```

160205 - 1

compiler data

number of instructions executed: 139858
compile time (microseconds): 553232.50
average instruction time: 3.96
object--program length: 208

execution data

number of instructions executed: 1230657
execution time (microseconds): 6346575.00
average instruction time: 5.16

profile

linenumber	count	time	%
3	71	356.25	0.0
5	85010	443773.75	7.0
6	205000	1036082.50	16.3
8	95	467.50	0.0
9	155	751.25	0.0
10	199788	1101747.50	17.4
11	481668	2463777.50	38.8
14	6	21.25	0.0
15	85010	443773.75	7.0
16	155000	779832.50	12.3
17	16016	62920.00	1.0
20	1923	8951.25	0.1

A.5 Pentomino

160205 - 1

```

1  _b_e_g_i_n _c_o_m_m_e_n_t pentomino, 130968;
2  _i_n_t_e_g_e_r score, nummer, lengte, breedte, aantal stenen, aantal standen,
3  i, j, k, teller, lb;
4  lengte:= read; breedte:= read; aantal stenen:= read; aantal standen:= read;
5  lb:= lengte * (breedte - 1);
6
7  _b_e_g_i_n _i_n_t_e_g_e_r _a_r_r_a_y bord [-39 : 100],
8  standen, wijzer [1 : aantal stenen],
9  informatie [1 : 8 * aantal standen];
10 _b_o_o_l_e_a_n _a_r_r_a_y ongebruikt [1 : aantal stenen];
11
12 _p_r_o_c_e_d_u_r_e output;
13 _b_e_g_i_n _i_n_t_e_g_e_r i, j;
14 score:= score + 1;
15 SPACE (1);
16 _f_o_r j:= 1 _s_t_e_p 1 _u_n_t_i_l 2 * lengte _d_o
17 _b_e_g_i_n PRINTTEXT (|<-|>); SPACE (1) _e_n_d ;
18 _c_o_m_m_e_n_t ABSFIXT (6, 2, time); NLCR;
19 _f_o_r i:= 0 _s_t_e_p lengte _u_n_t_i_l lb _d_o
20 _b_e_g_i_n PRINTTEXT (|<|>);
21 _f_o_r j:= 1 _s_t_e_p 1 _u_n_t_i_l lengte - 1 _d_o
22 _b_e_g_i_n SPACE (3);
23 _i_f bord [i + j] != bord [i + j + 1]
24 _t_h_e_n PRINTTEXT (|<I|>) _e_l_s_e SPACE (1)
25 _e_n_d ;
26 SPACE (3); PRINTTEXT (|<I|>); NLCR; SPACE (1);
27 _i_f i < lb
28 _t_h_e_n _b_e_g_i_n _f_o_r j:= 1 _s_t_e_p 1 _u_n_t_i_l lengte _d_o
29 _b_e_g_i_n _i_f bord [i + j] != bord [i + j + lengte]
30 _t_h_e_n _b_e_g_i_n PRINTTEXT (|<-|>); SPACE (1);
31 PRINTTEXT (|<-|>); SPACE (1)
32 _e_n_d
33 _e_l_s_e SPACE (4)
34 _e_n_d
35 _e_n_d
36 _e_l_s_e _f_o_r j:= 1 _s_t_e_p 1 _u_n_t_i_l 2 * lengte _d_o
37 _b_e_g_i_n PRINTTEXT (|<-|>); SPACE (1) _e_n_d ;
38 NLCR
39 _e_n_d ;
40 NLCR; NLCR;
41 _i_f score = 7 _t_h_e_n _g_o_t_o ex
42 _e_n_d output;
43
44 _p_r_o_c_e_d_u_r_e up (veld, kolom); _v_a_l_u_e veld, kolom;
45 _i_n_t_e_g_e_r veld, kolom;
46 _b_e_g_i_n _i_n_t_e_g_e_r i, j, k, r, w, steen, aantal;
47 nummer:= nummer + 1;
48 _f_o_r steen:= 1 _s_t_e_p 1 _u_n_t_i_l aantal stenen _d_o
49 _i_f ongebruikt [steen] _t_h_e_n
50 _b_e_g_i_n ongebruikt [steen]:= _f_a_l_s_e ;
51 bord [veld]:= steen;
52 aantal:= standen [steen] - 1;
53 _f_o_r i:= 0 _s_t_e_p 1 _u_n_t_i_l aantal _d_o
54 _b_e_g_i_n w:= wijzer [steen] + 4 * i;
55 _i_f bord [informatie [w] + veld] = 0 _t_h_e_n
56 _b_e_g_i_n _i_f bord [informatie [w + 1] + veld] = 0 _t_h_e_n
57 _b_e_g_i_n _i_f bord [informatie [w + 2] + veld] = 0 _t_h_e_n
58 _b_e_g_i_n _i_f bord [informatie [w + 3] + veld] = 0 _t_h_e_n
59 _b_e_g_i_n _f_o_r j:= 0, 1, 2, 3 _d_o

```

```

160205 -      1

60      bord [informatie [w + j] + veld]:= steen;
61      _i_f nummer = aantal stenen
62      _t_h_e_n output
63      _e_n_d
64      _b_e_g_i_n
65      _f_o_r k:= kolom _s_t_e_p 1 _u_n_t_i_l lengte _d_o
66      _f_o_r r:= 0 _s_t_e_p lengte _u_n_t_i_l lb _d_o
67      _i_f bord [r + k] = 0 _t_h_e_n _g_o_t_o beet;
68      beet: up (r + k, k)
69      _e_n_d ;
70      _f_o_r j:= 0, 1, 2, 3 _d_o
71      bord [informatie [w + j] + veld]:= 0
72      _e_n_d
73      _e_n_d
74      _e_n_d
75      _e_n_d
76      _e_n_d ;
77      ongebruikt [steen]:= _t_r_u_e
78      _e_n_d ;
79      bord [veld]:= 0;
80      down: nummer:= nummer - 1
81      _e_n_d up;
82
83      teller:= 1;
84      _f_o_r i:= 1 _s_t_e_p 1 _u_n_t_i_l aantal stenen _d_o
85      _b_e_g_i_n j:= read; _c_o_m_m_e_n_t steennummer, wordt niet gebruikt;
86      wijzer [i]:= teller; standen [i]:= read;
87      _f_o_r j:= 1 _s_t_e_p 1 _u_n_t_i_l standen [i] _d_o
88      _b_e_g_i_n _f_o_r k:= 0 _s_t_e_p 1 _u_n_t_i_l 3 _d_o
89      informatie [teller + k]:= read;
90      teller:= teller + 4
91      _e_n_d
92      _e_n_d ;
93      _f_o_r i:= - 39 _s_t_e_p 1 _u_n_t_i_l 0,
94      61 _s_t_e_p 1 _u_n_t_i_l 100 _d_o bord [i]:= - 1;
95      _f_o_r i:= 1 _s_t_e_p 1 _u_n_t_i_l 60 _d_o bord [i]:= 0;
96      _f_o_r i:= 1 _s_t_e_p 1 _u_n_t_i_l aantal stenen _d_o
97      ongebruikt [i]:= _t_r_u_e ;
98      score:= nummer:= 0;
99      NLCR; PRINTTEXT (|<The first 7 solutions:|>); NLCR; NLCR; NLCR;
100     up (1, 1);
101     ex:
102     _e_n_d
103     _e_n_d

```

160205 - 1

The first 7 solutions:

```

-----
I I I           I I           I
I I           I I           I I I I
I           I I           I I           I
-----
I I           I           I I I
I           I           I I           I
I I           I           I           I
-----

```

```

-----
I I I           I           I
I I           I           I           I
I           I I I           I           I
-----
I I           I I I I           I           I
I           I I           I           I I
I I           I           I           I
-----

```

```

-----
I I I           I           I
I I           I           I           I
I           I I I           I           I
-----
I I           I I I I           I           I
I           I I           I           I I
I I           I I           I           I
-----

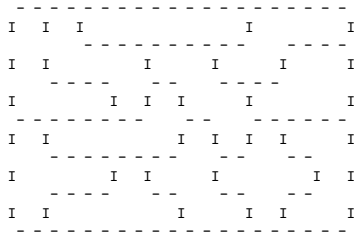
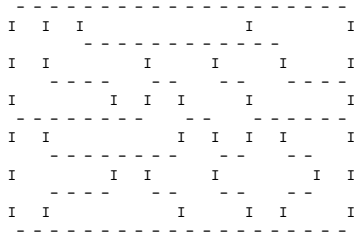
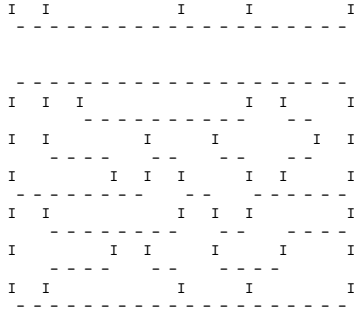
```

```

-----
I I I           I I           I
I I           I           I           I
I           I I I           I I           I
-----
I I           I I I           I I           I
I           I I           I           I
I I           I I           I           I
-----

```

160205 - 1



160205 - 1

compiler data
 number of instructions executed: 960488
 compile time (microseconds): 3747490.00
 average instruction time: 3.90
 object--program length: 1019

execution data
 number of instructions executed: 10813999
 execution time (microseconds): 51433663.75
 average instruction time: 4.76

profile

linenumber	count	time	%
4	3271	12918.75	0.0
5	10	58.75	0.0
7	180	790.00	0.0
8	68	308.75	0.0
10	72	361.25	0.0
14	49	192.50	0.0
15	476	2030.00	0.0
16	2646	16397.50	0.0
17	21280	87850.00	0.2
18	266	1146.25	0.0
19	833	4926.25	0.0
20	3612	14332.50	0.0
21	7560	42210.00	0.1
22	43848	189945.00	0.4
23	29292	128972.50	0.3
24	14703	58093.75	0.1
26	12684	53550.00	0.1
27	294	1417.50	0.0
28	6580	37406.25	0.1
29	14350	65187.50	0.1
30	32832	135540.00	0.3
31	33048	136080.00	0.3
33	18760	81070.00	0.2
36	2646	16397.50	0.0
37	21280	87850.00	0.2
38	1617	7001.25	0.0
40	518	2240.00	0.0
41	77	318.75	0.0
47	23471	92207.50	0.2
48	739675	4229616.25	8.2
49	1245270	6180237.50	12.0
50	380029	1919248.75	3.7
51	268801	1251315.00	2.4
52	203918	868968.75	1.7
53	798954	4573112.50	8.9
54	943525	4666692.50	9.1
55	1358676	5708326.25	11.1
56	557923	2337245.00	4.5
57	322788	1352220.00	2.6
58	191734	803210.00	1.6
59	144437	621415.00	1.2
60	618056	2838355.00	5.5
61	16795	67180.00	0.1
62	228	917.50	0.0
64	6704	25140.00	0.0
65	80390	427622.50	0.8
66	393921	2301293.75	4.5
67	461047	1969930.00	3.8
68	361994	1516697.50	2.9
70	143921	619195.00	1.2
71	615848	2761275.00	5.4
77	379537	1893615.00	3.7
79	96889	434330.00	0.8
80	40092	167050.00	0.3
83	6	21.25	0.0
84	221	1168.75	0.0
85	6363	25352.50	0.0
86	3946	16716.25	0.0
87	2304	10770.00	0.0

88	5100	25575.00	0.0
89	97268	401593.75	0.8
90	420	1650.00	0.0
93	1391	7232.50	0.0
94	2320	10200.00	0.0
95	2537	12146.25	0.0
96	221	1168.75	0.0
97	492	2431.25	0.0
98	7	26.25	0.0
99	1000	4271.25	0.0
100	90	355.00	0.0
102	103	471.25	0.0

A.6 Lisp interpreter

```

160205 -      1

1  _b_e_g_i_n  _c_o_m_m_e_n_t  ALGOL 60 version of program lisp(input,output).
2
3  ***      version 1, March 28, 1988      ***
4  ***      author: F.E.J. Kruseman Aretz      ***
5  *** Philips Research Laboratory Eindhoven ***;
6
7
8  _i_n_t_e_g_e_r  maxidf,maxnbr,maxstruct; maxidf:= 200; maxnbr:= 200; maxstruct:= 2000;
9
10
11  _b_e_g_i_n
12
13  _i_n_t_e_g_e_r  sym,shift,lastidf,lastnbr,d24,d25,free,indentation,linewidth,dummy,
14
15      f,
16      args,
17      p,
18      id,
19      olp,
20      t,
21      nilv,
22      quote,
23      cond,
24      lambda,
25      define,
26      car,
27      cdr,
28      cons,
29      equal,
30      atom,
31      numberp,
32      lessp,
33      greaterp,
34      add1,
35      sub1,
36      add,
37      minus,
38      timesv,
39      divf;
40
41  _i_n_t_e_g_e_r  _a_r_r_a_y  idf[0:maxidf,0:9],alist[0:maxidf];
42  _r_e_a_l      _a_r_r_a_y  nbr[0:maxnbr];
43  _i_n_t_e_g_e_r  _a_r_r_a_y  a,d[1:maxstruct];
44  _b_o_o_l_e_a_n  _a_r_r_a_y  m[1:maxstruct];
45
46
47  _c_o_m_m_e_n_t  *** error handling ***;
48
49  _p_r_o_c_e_d_u_r_e  errorhandler(errorstring); _s_t_r_i_n_g  errorstring;
50  _b_e_g_i_n  NLCR; NLCR; PRINTTEXT(|<+++ error: |>); PRINTTEXT(errorstring);
51  _g_o_t_o  ex;
52  _e_n_d  errorhandler;
53
54
55  _c_o_m_m_e_n_t  *** representation dependent functions ***;
56
57  _p_r_o_c_e_d_u_r_e  collect garbage;
58  _b_e_g_i_n  _i_n_t_e_g_e_r  i,j;
59      free:= 0;

```

```

160205 - 1

60  _f_o_r i:=1 _s_t_e_p 1 _u_n_t_i_l maxstruct _d_o m[i]:= _t_r_u_e;
61  NLCR; PRINTTEXT(|<garbage collector: |>);
62  mark(olp);
63  _f_o_r i:=0 _s_t_e_p 1 _u_n_t_i_l lastidf - 1 _d_o mark(alist[i]);
64  _f_o_r i:=1 _s_t_e_p 1 _u_n_t_i_l maxstruct _d_o
65  _i_f m[i] _t_h_e_n _b_e_g_i_n a[i]:= free; free:= i _e_n_d;
66  _i_f free = 0 _t_h_e_n errorhandler(|<free list exhausted|>);
67  i:= i + 1; j:= free;
68  _f_o_r j:= carf(j) _w_h_i_l_e j != 0 _d_o i:= i + 1;
69  ABSFIXT(4,0,i); NLCR
70  _e_n_d collect garbage;
71
72  _p_r_o_c_e_d_u_r_e mark(ref); _v_a_l_u_e ref; _i_n_t_e_g_e_r ref;
73  _b_e_g_i_n
74  work: _i_f ref < d24
75  _t_h_e_n _b_e_g_i_n _i_f m[ref]
76  _t_h_e_n _b_e_g_i_n m[ref]:= _f_a_l_s_e;
77  mark(a[ref]); ref:= d[ref]; _g_o_t_o work
78  _e_n_d
79  _e_n_d
80  _e_n_d mark;
81
82  _i_n_t_e_g_e_r _p_r_o_c_e_d_u_r_e createidf;
83  _b_e_g_i_n _i_n_t_e_g_e_r i,j;
84  i:= 0;
85  _f_o_r dummy:= 0 _w_h_i_l_e i < lastidf _d_o
86  _b_e_g_i_n _f_o_r j:= 0 _s_t_e_p 1 _u_n_t_i_l 9 _d_o
87  _i_f idf[lastidf,j] != idf[i,j] _t_h_e_n _g_o_t_o diff;
88  _g_o_t_o old;
89  diff: i:= i + 1
90  _e_n_d;
91  new: i:= lastidf; alist[i]:= nilv; lastidf:= lastidf + 1;
92  _i_f lastidf = maxidf _t_h_e_n
93  _b_e_g_i_n _f_o_r i:= 0 _s_t_e_p 1 _u_n_t_i_l 99 _d_o
94  _b_e_g_i_n NLCR; write(d25+i) _e_n_d;
95  errorhandler(|<too much identifiers|>)
96  _e_n_d;
97  old: createidf:= d25 + i
98  _e_n_d createidf;
99
100 _i_n_t_e_g_e_r _p_r_o_c_e_d_u_r_e createnum(x); _r_e_a_l x;
101 _b_e_g_i_n _i_n_t_e_g_e_r i;
102 nbr[last nbr]:= x; i:= 0;
103 _f_o_r dummy:= 0 _w_h_i_l_e i < last nbr _d_o
104 _b_e_g_i_n _i_f nbr[last nbr] = nbr[i] _t_h_e_n _g_o_t_o old;
105 i:= i + 1
106 _e_n_d;
107 new: i:= last nbr; last nbr:= last nbr + 1;
108 _i_f last nbr = maxnbr _t_h_e_n errorhandler(|<too much numbers|>);
109 old: createnum:= d24 + i
110 _e_n_d createnum;
111
112 _b_o_o_l_e_a_n _p_r_o_c_e_d_u_r_e atomf(x); _v_a_l_u_e x; _i_n_t_e_g_e_r x;
113 _b_e_g_i_n atomf:= x > d24 _e_n_d atomf;
114
115 _b_o_o_l_e_a_n _p_r_o_c_e_d_u_r_e numberpf(x); _v_a_l_u_e x; _i_n_t_e_g_e_r x;
116 _b_e_g_i_n numberpf:= x > d24 ^ x < d25 _e_n_d numberpf;
117
118 _p_r_o_c_e_d_u_r_e getidfval(x,idf); _v_a_l_u_e x; _i_n_t_e_g_e_r x,idf;
119 _b_e_g_i_n idf:= x - d25 _e_n_d getidfval;

```

```

160205 - 1

120
121 _r_e_a_l _p_r_o_c_e_d_u_r_e numval(x); _v_a_l_u_e x; _i_n_t_e_g_e_r x;
122 _b_e_g_i_n numval:=nbr[ x - d24] _e_n_d numval;
123
124 _i_n_t_e_g_e_r _p_r_o_c_e_d_u_r_e carf(x); _v_a_l_u_e x; _i_n_t_e_g_e_r x;
125 _b_e_g_i_n _i_f x > d24
126 _t_h_e_n errorhandler(|<car undefined for atomic lisp value|>);
127 carf:= a[x]
128 _e_n_d carf;
129
130 _i_n_t_e_g_e_r _p_r_o_c_e_d_u_r_e cdrf(x); _v_a_l_u_e x; _i_n_t_e_g_e_r x;
131 _b_e_g_i_n _i_f x > d24
132 _t_h_e_n errorhandler(|<cdr undefined for atomic lisp value|>);
133 cdrf:= d[x]
134 _e_n_d cdrf;
135
136 _i_n_t_e_g_e_r _p_r_o_c_e_d_u_r_e consf(x,y);
137 _v_a_l_u_e x,y; _i_n_t_e_g_e_r x,y;
138 _b_e_g_i_n _i_n_t_e_g_e_r n;
139 _i_f free = 0 _t_h_e_n collect garbage;
140 n:= free; free:= a[free];
141 a[n]:= x; d[n]:= y; consf:= n
142 _e_n_d consf;
143
144 _p_r_o_c_e_d_u_r_e returncell(x); _v_a_l_u_e x; _i_n_t_e_g_e_r x;
145 _b_e_g_i_n a[x]:= free; free:= x _e_n_d;
146
147 _p_r_o_c_e_d_u_r_e returnlist(x); _v_a_l_u_e x; _i_n_t_e_g_e_r x;
148 _b_e_g_i_n _f_o_r dummy:= 0 _w_h_i_l_e x != nilv _d_o
149 _b_e_g_i_n returncell(x); x:= d[x] _e_n_d
150 _e_n_d returnlist;
151
152 _p_r_o_c_e_d_u_r_e recycle(x); _v_a_l_u_e x; _i_n_t_e_g_e_r x;
153 _b_e_g_i_n _f_o_r dummy:= 0 _w_h_i_l_e ~ atomf(x) _d_o
154 _b_e_g_i_n recycle(a[x]); returncell(x); x:= d[x] _e_n_d
155 _e_n_d recycle;
156
157 _b_o_o_l_e_a_n _p_r_o_c_e_d_u_r_e equalf(x,y);
158 _v_a_l_u_e x,y; _i_n_t_e_g_e_r x,y;
159 _b_e_g_i_n _s_w_i_t_c_h s:= str,num,id;
160 work:
161 _i_f x _: d24 = y _: d24
162 _t_h_e_n _b_e_g_i_n _g_o_t_o s[x _: d24 + 1];
163 id: num: equalf:= x = y; _g_o_t_o ex;
164 str: _i_f equalf(a[x],a[y])
165 _t_h_e_n _b_e_g_i_n x:= d[x]; y:= d[y];
166 _g_o_t_o work
167 _e_n_d
168 _e_l_s_e equalf:= _f_a_l_s_e
169 _e_n_d
170 _e_l_s_e equalf := _f_a_l_s_e;
171 ex:
172 _e_n_d equalf;
173
174
175
176 _c_o_m_m_e_n_t *** input procedures ***;
177
178 _i_n_t_e_g_e_r _p_r_o_c_e_d_u_r_e RESYM;
179 _b_e_g_i_n _i_n_t_e_g_e_r s;

```

```

160205 -      1
180  s:= read;
181  _if s = 122 # s = 124      _t_h_e_n _b_e_g_i_n shift:= s;
182                                RESYM:= RESYM
183                                _e_n_d _e_l_s_e
184  _if s = 16                _t_h_e_n RESYM:= 93 _e_l_s_e
185  _if s = 26                _t_h_e_n RESYM:= 119 _e_l_s_e
186  _if s = 8 ^ shift = 124  _t_h_e_n RESYM:= 98 _e_l_s_e
187  _if s = 25 ^ shift = 124 _t_h_e_n RESYM:= 99 _e_l_s_e
188  _if s = 107               _t_h_e_n RESYM:= 88 _e_l_s_e
189  _if s = 32                _t_h_e_n RESYM:= 0  _e_l_s_e
190  _b_e_g_i_n s:= s:32*32 + s - s_:16*16;
191  _if s = 0 _t_h_e_n errorhandler(|<eof>|);
192  RESYM:= _if s < 10 _t_h_e_n s      _e_l_s_e
193          _if s < 64 _t_h_e_n s - 6 _e_l_s_e
194          _if s < 96 _t_h_e_n s - 46 _e_l_s_e s - 87
195  _e_n_d
196  _e_n_d RESYM;
197
198
199
200
201  _p_r_o_c_e_d_u_r_e nextsym;
202  _b_e_g_i_n sym:= RESYM; PRSYM(sym) _e_n_d nextsym;
203
204  _p_r_o_c_e_d_u_r_e skipspace;
205  _b_e_g_i_n _f_o_r dummy:= 0
206  _w_h_i_l_e sym = 93 # sym = 118 # sym = 119 _d_o nextsym
207  _e_n_d skipspace;
208
209  _i_n_t_e_g_e_r _p_r_o_c_e_d_u_r_e number;
210  _b_e_g_i_n _r_e_a_l x; _b_o_o_l_e_a_n signed;
211  x:= 0; signed:= (sym = 65);
212  _if signed
213  _t_h_e_n _b_e_g_i_n nextsym;
214  _if sym > 9 _t_h_e_n errorhandler(|<digit expected in input>|)
215  _e_n_d;
216  _f_o_r dummy:= 0 _w_h_i_l_e sym < 10 _d_o
217  _b_e_g_i_n x:= 10 * x + sym; nextsym _e_n_d;
218  number:= createnum(_if signed _t_h_e_n -x _e_l_s_e x)
219  _e_n_d number;
220
221  _i_n_t_e_g_e_r _p_r_o_c_e_d_u_r_e identifier;
222  _b_e_g_i_n _i_n_t_e_g_e_r i;
223  idf[lastidf,0]:= sym; nextsym;
224  _f_o_r i:= 1 _s_t_e_p 1 _u_n_t_i_l 9 _d_o idf[lastidf,i]:= 93;
225  i:= 0;
226  _f_o_r dummy:= 0 _w_h_i_l_e sym < 64 ^ i < 9 _d_o
227  _b_e_g_i_n i:= i + 1; idf[lastidf,i]:= sym; nextsym _e_n_d;
228  _f_o_r dummy:= 0 _w_h_i_l_e sym < 64 _d_o nextsym;
229  identifier:= createidf
230  _e_n_d identifier;
231
232  _i_n_t_e_g_e_r _p_r_o_c_e_d_u_r_e nextitem;
233  _b_e_g_i_n _i_n_t_e_g_e_r lv,op;
234  skipspace;
235  _if sym < 10 # sym = 65 _t_h_e_n nextitem:= number
236  _e_l_s_e
237  _if sym < 64 _t_h_e_n nextitem := identifier
238  _e_l_s_e
239  _if sym = 98

```

```

160205 - 1

240  _t_h_e_n _b_e_g_i_n nextsym; skipspace;
241  _i_f sym = 99
242  _t_h_e_n _b_e_g_i_n nextitem:= nilv; nextsym _e_n_d
243  _e_l_s_e _b_e_g_i_n op:= olp; olp:= consf(nilv,olp);
244  lv:= a[olp]:= consf(nilv,nilv); nextitem:= lv;
245  a[lv]:= nextitem; skipspace;
246  _i_f sym = 88
247  _t_h_e_n _b_e_g_i_n nextsym; d[lv]:= nextitem;
248  skipspace;
249  _i_f sym != 99
250  _t_h_e_n errorhandler
251  (|<close missing for dotted pair in input|>)
252  _e_n_d
253  _e_l_s_e _f_o_r dummy:= 0 _w_h_i_l_e sym != 99 _d_o
254  _b_e_g_i_n lv:= d[lv]:= consf(nilv,nilv);
255  a[lv]:= nextitem; skipspace
256  _e_n_d;
257  nextsym;
258  olp:= op
259  _e_n_d;
260  _e_n_d
261  _e_l_s_e
262  _i_f sym = 120
263  _t_h_e_n _b_e_g_i_n nextsym;
264  op:= olp; olp:= consf(nilv,olp);
265  lv:= a[olp]:= consf(nilv,nilv); nextitem:= lv;
266  a[lv]:= quote; lv:= d[lv]:= consf(nilv,nilv); a[lv]:= nextitem;
267  olp:= op
268  _e_n_d
269  _e_l_s_e errorhandler(|illegal symbol in input|>)
270  _e_n_d nextitem;
271
272
273
274  _c_o_m_m_e_n_t *** output procedures ***;
275
276  _p_r_o_c_e_d_u_r_e PRSYM(sym); _v_a_l_u_e sym; _i_n_t_e_g_e_r sym;
277  _b_e_g_i_n _s_w_i_t_c_h sw:= a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,
278  a,b,c,d,e,f,g,h,i,j,k,l,m,
279  n,o,p,q,r,s,t,u,v,w,x,y,z;
280  _i_f sym = 93 _t_h_e_n SPACE(1) _e_l_s_e
281  _i_f sym = 88 _t_h_e_n PRINTTEXT(|<|>) _e_l_s_e
282  _i_f sym = 98 _t_h_e_n PRINTTEXT(|<|>) _e_l_s_e
283  _i_f sym = 99 _t_h_e_n PRINTTEXT(|<|>) _e_l_s_e
284  _i_f sym = 119 _t_h_e_n NLCR _e_l_s_e
285  _b_e_g_i_n _i_f sym > 35
286  _t_h_e_n errorhandler(|illegal output symbol|>);
287  _g_o_t_o sw[sw+1];
288  a0: PRINTTEXT(|<0|>); _g_o_t_o ex;
289  a1: PRINTTEXT(|<1|>); _g_o_t_o ex;
290  a2: PRINTTEXT(|<2|>); _g_o_t_o ex;
291  a3: PRINTTEXT(|<3|>); _g_o_t_o ex;
292  a4: PRINTTEXT(|<4|>); _g_o_t_o ex;
293  a5: PRINTTEXT(|<5|>); _g_o_t_o ex;
294  a6: PRINTTEXT(|<6|>); _g_o_t_o ex;
295  a7: PRINTTEXT(|<7|>); _g_o_t_o ex;
296  a8: PRINTTEXT(|<8|>); _g_o_t_o ex;
297  a9: PRINTTEXT(|<9|>); _g_o_t_o ex;
298  a: PRINTTEXT(|<a|>); _g_o_t_o ex;
299  b: PRINTTEXT(|<b|>); _g_o_t_o ex;

```

```

160205 - 1
300 c: PRINTTEXT(|<c|>); _g_o_t_o ex;
301 d: PRINTTEXT(|<d|>); _g_o_t_o ex;
302 e: PRINTTEXT(|<e|>); _g_o_t_o ex;
303 f: PRINTTEXT(|<f|>); _g_o_t_o ex;
304 g: PRINTTEXT(|<g|>); _g_o_t_o ex;
305 h: PRINTTEXT(|<h|>); _g_o_t_o ex;
306 i: PRINTTEXT(|<i|>); _g_o_t_o ex;
307 j: PRINTTEXT(|<j|>); _g_o_t_o ex;
308 k: PRINTTEXT(|<k|>); _g_o_t_o ex;
309 l: PRINTTEXT(|<l|>); _g_o_t_o ex;
310 m: PRINTTEXT(|<m|>); _g_o_t_o ex;
311 n: PRINTTEXT(|<n|>); _g_o_t_o ex;
312 o: PRINTTEXT(|<o|>); _g_o_t_o ex;
313 p: PRINTTEXT(|<p|>); _g_o_t_o ex;
314 q: PRINTTEXT(|<q|>); _g_o_t_o ex;
315 r: PRINTTEXT(|<r|>); _g_o_t_o ex;
316 s: PRINTTEXT(|<s|>); _g_o_t_o ex;
317 t: PRINTTEXT(|<t|>); _g_o_t_o ex;
318 u: PRINTTEXT(|<u|>); _g_o_t_o ex;
319 v: PRINTTEXT(|<v|>); _g_o_t_o ex;
320 w: PRINTTEXT(|<w|>); _g_o_t_o ex;
321 x: PRINTTEXT(|<x|>); _g_o_t_o ex;
322 y: PRINTTEXT(|<y|>); _g_o_t_o ex;
323 z: PRINTTEXT(|<z|>);
324 _e_n_d;
325 ex:
326 _e_n_d PRSYM;
327
328 _p_r_o_c_e_d_u_r_e analyse(x,r); _v_a_l_l_u_e x; _i_n_t_e_g_e_r x,r;
329 _b_e_g_i_n _i_n_t_e_g_e_r n,l; _b_o_o_l_e_a_n simple;
330 _i_f numberpf(x)
331 _t_h_e_n _b_e_g_i_n _r_e_a_l dg,v,absv;
332 v:= numval(x);
333 _i_f v > 0
334 _t_h_e_n _b_e_g_i_n absv:= v; l:= 1 _e_n_d
335 _e_l_s_e _b_e_g_i_n absv:= - v;l:= 2 _e_n_d;
336 dg:= 10;
337 _f_o_r dummy:= 0 _w_h_i_l_e dg < absv _d_o
338 _b_e_g_i_n l:= l + 1; dg:= 10 * dg _e_n_d;
339 r:= createnum(l)
340 _e_n_d
341 _e_l_s_e
342 _i_f atomf(x)
343 _t_h_e_n _b_e_g_i_n getidfval(x,id); n:= 10;
344 _f_o_r dummy:= 0 _w_h_i_l_e idf[id,n-1] = 93 _d_o n:= n - 1;
345 r:= createnum(n)
346 _e_n_d
347 _e_l_s_e
348 _i_f islist(x)
349 _t_h_e_n _b_e_g_i_n indentation:= indentation + 1;
350 analyselist(x,r,l,simple);
351 indentation:= indentation - 1;
352 _i_f simple ^ indentation + 1 < linewidth
353 _t_h_e_n _b_e_g_i_n recycle(r); r:= createnum(l) _e_n_d
354 _e_n_d
355 _e_l_s_e _b_e_g_i_n indentation:= indentation + 1;
356 olp:= consf(nilv,olp);
357 r:= a[olp]:= consf(nilv,nilv);
358 analyse(carf(x),a[r]); analyse(cdrf(x),d[r]);
359 indentation:= indentation - 1;

```



```

160205 - 1

360      _i_f atomf(a[r]) ^ atomf(d[r])
361      _t_h_e_n _b_e_g_i_n l:= numval(carf(r)) + numval(cdrf(x)) + 5;
362      _i_f indentation + l < linewidth
363      _t_h_e_n _b_e_g_i_n recycle(r); r:= createnum(l) _e_n_d
364      _e_n_d;
365      returncell(olp); olp:= d[olp]
366      _e_n_d
367 _e_n_d analyse;
368
369 _p_r_o_c_e_d_u_r_e analyselist(x,r,l,simple);
370 _v_a_l_u_e x; _i_n_t_e_g_e_r x,r,l; _b_o_o_l_e_a_n simple;
371 _b_e_g_i_n _i_f x = nilv
372 _t_h_e_n _b_e_g_i_n r:= nilv; l:= 1; simple:= _t_r_u_e _e_n_d
373 _e_l_s_e _b_e_g_i_n olp:= consf(nilv, olp);
374 r:= a[olp]:= consf(nilv,nilv);
375 analyse(carf(x),a[r]); analyselist(cdrf(x),d[r],l,simple);
376 _i_f simple ^ atomf(a[r])
377 _t_h_e_n l:= numval(a[r]) + 1 + 1
378 _e_l_s_e simple:= _f_a_l_s_e;
379 returncell(olp); olp:= d[olp]
380 _e_n_d
381 _e_n_d analyselist;
382
383 _b_o_o_l_e_a_n _p_r_o_c_e_d_u_r_e islist(x); _v_a_l_u_e x; _i_n_t_e_g_e_r x;
384 _b_e_g_i_n
385 work: _i_f atomf(x)
386 _t_h_e_n islist:= equalf(x,nilv)
387 _e_l_s_e _b_e_g_i_n x:= cdrf(x); _g_o_t_o work _e_n_d
388 _e_n_d islist;
389
390 _p_r_o_c_e_d_u_r_e writenumber(x); _v_a_l_u_e x; _i_n_t_e_g_e_r x;
391 _b_e_g_i_n _i_n_t_e_g_e_r n,d,v;
392 v:= numval(x);
393 _i_f v < 0 _t_h_e_n v:= - v;
394 d:= 10;
395 _f_o_r dummy:= 0 _w_h_i_l_e d < v _d_o d:= d * 10;
396 _f_o_r d:= d : 10 _w_h_i_l_e d > 0.5 _d_o
397 _b_e_g_i_n n:= v : d; PRSYM(n); v:= v - d * n _e_n_d
398 _e_n_d writenumber;
399
400 _p_r_o_c_e_d_u_r_e writeidentifier(x); _v_a_l_u_e x; _i_n_t_e_g_e_r x;
401 _b_e_g_i_n _i_n_t_e_g_e_r i;
402 getidfval(x,id);
403 _f_o_r i:= 0 _s_t_e_p 1 _u_n_t_i_l 9 _d_o
404 _i_f idf[id,i] != 93 _t_h_e_n PRSYM(idf[id,i])
405 _e_n_d writeidentifier;
406
407 _p_r_o_c_e_d_u_r_e writelist(x,r); _v_a_l_u_e x,r; _i_n_t_e_g_e_r x,r;
408 _b_e_g_i_n _i_n_t_e_g_e_r a,ind; _b_o_o_l_e_a_n simple,nl;
409 PRSYM(98);
410 _i_f atomf(r)
411 _t_h_e_n _b_e_g_i_n _f_o_r dummy:= 0 _w_h_i_l_e x != nilv _d_o
412 _b_e_g_i_n writevalue(carf(x),r); x:= cdrf(x);
413 _i_f x != nilv _t_h_e_n PRSYM(93)
414 _e_n_d
415 _e_n_d
416 _e_l_s_e _b_e_g_i_n indentation:= indentation + 1; ind:= indentation;
417 _f_o_r dummy:= 0 _w_h_i_l_e x != nilv _d_o
418 _b_e_g_i_n a:= carf(r);
419 simple:= atomf(a);

```

```

160205 -      1

420         _i_f simple
421         _t_h_e_n nl:= numval(a) + indentation > linewidth
422         _e_l_s_e nl:= indentation > ind;
423         _i_f nl
424         _t_h_e_n _b_e_g_i_n indentation:= ind;
425                 NLCR; SPACE(ind)
426                 _e_n_d
427         _e_l_s_e _i_f indentation > ind _t_h_e_n PRSYM(93);
428         writevalue(carf(x),a);
429         _i_f simple
430         _t_h_e_n indentation:= indentation + numval(a) + 1
431         _e_l_s_e indentation:= linewidth + 1;
432         x:= cdrf(x); r:= cdrf(r)
433         _e_n_d;
434         indentation:= ind - 1; NLCR; SPACE(indentation)
435         _e_n_d;
436     PRSYM(99)
437 _e_n_d writelist;
438
439 _p_r_o_c_e_d_u_r_e writepair(x,r); _v_a_l_u_e x,r; _i_n_t_e_g_e_r x,r;
440 _b_e_g_i_n PRSYM(98);
441 _i_f atomf(r)
442 _t_h_e_n _b_e_g_i_n writevalue(carf(x),r); PRINTTEXT(|< . |>);
443         writevalue(cdrf(x),r)
444         _e_n_d
445 _e_l_s_e _b_e_g_i_n indentation:= indentation + 1;
446         writevalue(carf(x),carf(r));
447         NLCR; SPACE(indentation-1); PRINTTEXT(|< . |>);
448         NLCR; SPACE(indentation); writevalue(cdrf(x),cdrf(r));
449         NLCR; SPACE(indentation)
450         _e_n_d;
451     PRSYM(99)
452 _e_n_d writepair;
453
454 _p_r_o_c_e_d_u_r_e writevalue(x,r); _v_a_l_u_e x,r; _i_n_t_e_g_e_r x,r;
455 _b_e_g_i_n _i_f numberpf(x) _t_h_e_n writenumber(x)
456 _e_l_s_e
457 _i_f atomf(x) _t_h_e_n writeidentifier(x)
458 _e_l_s_e
459 _i_f islist(x) _t_h_e_n writelist(x,r)
460 _e_l_s_e writepair(x,r)
461 _e_n_d writevalue;
462
463 _p_r_o_c_e_d_u_r_e write(x); _v_a_l_u_e x; _i_n_t_e_g_e_r x;
464 _b_e_g_i_n _i_n_t_e_g_e_r r;
465     indentation:= 0;
466     analyse(x,r); writevalue(x,r); recycle(r)
467 _e_n_d write;
468
469
470
471 _c_o_m_m_e_n_t *** interpreter proper ***;
472
473
474 _i_n_t_e_g_e_r _p_r_o_c_e_d_u_r_e assoc(x); _v_a_l_u_e x; _i_n_t_e_g_e_r x;
475 _b_e_g_i_n _i_n_t_e_g_e_r ax;
476     getidfval(x,id); ax:= alist[id];
477     _i_f ax = nilv _t_h_e_n errorhandler(|<identifier has no value|>);
478     assoc:= carf(ax)
479 _e_n_d assoc;

```

```

160205 - 1

480
481 _p_r_o_c_e_d_u_r_e pairlis(x,y); _v_a_l_u_e x,y; _i_n_t_e_g_e_r x,y;
482 _b_e_g_i_n _f_o_r dummy:= 0 _w_h_i_l_e ~ equalf(x,nilv) _d_o
483   _b_e_g_i_n getidfval(carf(x),id); alist[id]:= consf(carf(y),alist[id]);
484   x:= cdrf(x); y:= cdrf(y)
485 _e_n_d
486 _e_n_d pairlis;
487
488 _p_r_o_c_e_d_u_r_e depairlis(x); _v_a_l_u_e x; _i_n_t_e_g_e_r x;
489 _b_e_g_i_n _f_o_r dummy:= 0 _w_h_i_l_e ~ equalf(x,nilv) _d_o
490   _b_e_g_i_n getidfval(carf(x),id); alist[id]:= cdrf(alist[id]);
491   x:= cdrf(x)
492 _e_n_d
493 _e_n_d depairlis;
494
495 _i_n_t_e_g_e_r _p_r_o_c_e_d_u_r_e eval(e); _v_a_l_u_e e; _i_n_t_e_g_e_r e;
496 _b_e_g_i_n _i_n_t_e_g_e_r care;
497 work: _i_f atomf(e)
498   _t_h_e_n _b_e_g_i_n _i_f equalf(e,nilv) # equalf(e,t) # numberpf(e)
499   _t_h_e_n eval:= e _e_l_s_e eval:= assoc(e)
500   _e_n_d
501   _e_l_s_e _b_e_g_i_n care:= carf(e);
502   _i_f equalf(care,cond)
503     _t_h_e_n _b_e_g_i_n e:= evcon(cdrf(e)); _g_o_t_o work _e_n_d
504   _e_l_s_e _i_f equalf(care,quote)
505     _t_h_e_n eval:= carf(cdrf(e))
506   _e_l_s_e _b_e_g_i_n olp:= consf(nilv,olp);
507     a[olp]:= evlist(cdrf(e)); eval:= apply(care,a[olp]);
508     returnlist(a[olp]); returncell(olp); olp:= cdrf(olp)
509     _e_n_d
510   _e_n_d
511 _e_n_d eval;
512
513 _i_n_t_e_g_e_r _p_r_o_c_e_d_u_r_e apply(f,x);
514 _v_a_l_u_e f,x; _i_n_t_e_g_e_r f,x;
515 _b_e_g_i_n
516 work: _i_f atomf(f)
517   _t_h_e_n _b_e_g_i_n
518     _i_f equalf(f,car) _t_h_e_n apply:= carf(carf(x))
519     _e_l_s_e
520     _i_f equalf(f,cdr) _t_h_e_n apply:= cdrf(carf(x))
521     _e_l_s_e
522     _i_f equalf(f,cons) _t_h_e_n apply:= consf(carf(x),carf(cdrf(x)))
523     _e_l_s_e
524     _i_f equalf(f,equal)
525     _t_h_e_n _b_e_g_i_n _i_f equalf(carf(x),carf(cdrf(x)))
526     _t_h_e_n apply:= t
527     _e_l_s_e apply:= nilv
528     _e_n_d
529   _e_l_s_e
530   _i_f equalf(f,atom) _t_h_e_n _b_e_g_i_n _i_f atomf(carf(x))
531     _t_h_e_n apply:= t
532     _e_l_s_e apply:= nilv
533     _e_n_d
534   _e_l_s_e
535   _i_f equalf(f,numberp) _t_h_e_n _b_e_g_i_n _i_f numberpf(carf(x))
536     _t_h_e_n apply:= t
537     _e_l_s_e apply:= nilv
538   _e_n_d
539   _e_l_s_e

```

```

160205 -      1

540      _i_f equalf(f,lessp)
541      _t_h_e_n _b_e_g_i_n _i_f numval(carf(x)) < numval(carf(cdrf(x)))
542      _t_h_e_n _t_h_e_n apply:= t
543      _e_l_s_e _e_l_s_e apply:= nilv
544      _e_n_d
545      _e_l_s_e
546      _i_f equalf(f,greaterp)
547      _t_h_e_n _b_e_g_i_n _i_f numval(carf(x)) > numval(carf(cdrf(x)))
548      _t_h_e_n _t_h_e_n apply:= t
549      _e_l_s_e _e_l_s_e apply:= nilv
550      _e_n_d
551      _e_l_s_e
552      _i_f equalf(f,add)
553      _t_h_e_n apply:= createnum(numval(carf(x)) + 1)
554      _e_l_s_e
555      _i_f equalf(f,sub1)
556      _t_h_e_n apply:= createnum(numval(carf(x)) - 1)
557      _e_l_s_e
558      _i_f equalf(f,add)
559      _t_h_e_n apply:= createnum(numval(carf(x)) + numval(carf(cdrf(x))))
560      _e_l_s_e
561      _i_f equalf(f,minus)
562      _t_h_e_n apply:= createnum(numval(carf(x)) - numval(carf(cdrf(x))))
563      _e_l_s_e
564      _i_f equalf(f,timesv)
565      _t_h_e_n apply:= createnum(numval(carf(x)) * numval(carf(cdrf(x))))
566      _e_l_s_e
567      _i_f equalf(f,divf)
568      _t_h_e_n apply:= createnum(numval(carf(x)) / numval(carf(cdrf(x))))
569      _e_l_s_e _b_e_g_i_n f:= assoc(f); _g_o_t_o work _e_n_d
570      _e_n_d
571      _e_l_s_e _b_e_g_i_n pairlis(carf(cdrf(f)),x);
572      apply:= eval(carf(cdrf(cdrf(f))));
573      depairlis(carf(cdrf(f)))
574      _e_n_d
575      _e_n_d apply;
576
577      _i_n_t_e_g_e_r _p_r_o_c_e_d_u_r_e evcon(x); _v_a_l_u_e x; _i_n_t_e_g_e_r x;
578      _b_e_g_i_n _i_n_t_e_g_e_r r;
579      work: r:= carf(x);
580      _i_f equalf(eval(carf(r)),nilv)
581      _t_h_e_n _b_e_g_i_n x:= cdrf(x); _g_o_t_o work _e_n_d
582      _e_l_s_e evcon:= carf(cdrf(r))
583      _e_n_d evcon;
584
585      _i_n_t_e_g_e_r _p_r_o_c_e_d_u_r_e evlist(x); _v_a_l_u_e x; _i_n_t_e_g_e_r x;
586      _b_e_g_i_n _i_n_t_e_g_e_r res;
587      _i_f equalf(x,nilv)
588      _t_h_e_n evlist:= nilv
589      _e_l_s_e _b_e_g_i_n olp:= consf(nilv,olp); a[olp]:= res:= consf(nilv,nilv);
590      a[res]:= eval(carf(x)); d[res]:= evlist(cdrf(x));
591      evlist:= res;
592      returncell(olp); olp:= cdrf(olp)
593      _e_n_d
594      _e_n_d evlist;
595
596
597
598      _c_o_m_m_e_n_t *** initialization ***;
599

```

```

160205 - 1

600 _p_r_o_c_e_d_u_r_e create(lv); _i_n_t_e_g_e_r lv;
601 _b_e_g_i_n skipspaces;
602 lv:= identifier;
603 _e_n_d create;
604
605 _p_r_o_c_e_d_u_r_e init;
606 _b_e_g_i_n _i_n_t_e_g_e_r i,j;
607 d24:= 16777216; d25:= 33554432;
608 last idf:= 0; sym:= 93; nilv:= d25 + 1;
609 create(t); create(nilv);
610 create(quote); create(cond);
611 create(lambda); create(define);
612 create(car); create(cdr);
613 create(cons); create(equal);
614 create(atom); create(numberp);
615 create(lessp); create(greaterp);
616 create(add1); create(sub1);
617 create(add); create(minus);
618 create(timesv); create(divf);
619
620 olp:= nilv;
621
622 free:= 1; last nbr:= 0; linewidth:= 40;
623 _f_o_r i:= 1 _s_t_e_p 1 _u_n_t_i_l maxstruct - 1 _d_o a[i]:= i + 1;
624 a[maxstruct]:= 0
625 _e_n_d init;
626
627
628
629 _c_o_m_m_e_n_t *** main program ***;
630
631
632 _p_r_o_c_e_d_u_r_e function definitions(x,a,r); _v_a_l_u_e x; _i_n_t_e_g_e_r x,a,r;
633 _b_e_g_i_n _i_n_t_e_g_e_r carx,lr;
634 _i_f equalf(x,nilv)
635 _t_h_e_n r:= nilv
636 _e_l_s_e _b_e_g_i_n carx:= carf(x);
637 a:= consf(consf(carf(carx),carf(cdrf(carx))),a);
638 function definitions(cdrf(x),a,lr);
639 r:= consf(carf(carx),lr)
640 _e_n_d
641 _e_n_d function definitions;
642
643 PRINTTEXT(|<Lisp interpreter version 1, Oktober 2004|>);
644 NLCR; NLCR;
645 init;
646 _f_o_r dummy:= 0 _w_h_i_l_e _t_r_u_e _d_o
647 _b_e_g_i_n olp:= consf(nilv,olp); a[olp]:= p:= consf(nilv,nilv);
648 a[p]:= f:= nextitem; d[p]:= args:= nextitem;
649 NLCR;
650 _i_f equalf(f,define)
651 _t_h_e_n _b_e_g_i_n args:= carf(args); PRSYM(98);
652 _f_o_r dummy:= 0 _w_h_i_l_e ~ equalf(args,nilv) _d_o
653 _b_e_g_i_n p:= carf(args); write(carf(p));
654 getidfval(carf(p),id);
655 alist[id]:= consf(carf(cdrf(p)),nilv);
656 args:= cdrf(args);
657 _i_f ~ equalf(args,nilv) _t_h_e_n SPACE(1)
658 _e_n_d;
659 PRSYM(99)

```

```
160205 -      1
660          _e_n_d
661      _e_l_s_e _b_e_g_i_n p:= apply(f,args);
662          NLCR; write(p)
663          _e_n_d;
664      olp:= cdrf(olp)
665      _e_n_d;
666  ex:
667  _e_n_d
668  _e_n_d
669
```

160205 - 1

Lisp interpreter version 1, Oktober 2004

```
t nil quote cond lambda define car cdr cons equal atom numberp
lessp greaterp add1 sub1 add minus times div
```

```
define ((
  (crossriver (lambda ( ) (complete (cons (i) nil))))
  (complete
    (lambda (path)
      (cond ((equal (car path) (f)) (cons path nil))
            (t (try path (fullmoveset))))
    ) ) )
  (try
    (lambda (path moveset)
      (cond ((null moveset) nil)
            ((feasible (car moveset) (car path))
             (append (try1 path (result (car moveset) (car path)))
                     (try path (cdr moveset))))
            (t (try path (cdr moveset))))
    ) ) )
  (try1
    (lambda (path newstate)
      (cond ((not (admissible newstate)) nil)
            ((member newstate path) nil)
            (t (complete (cons newstate path))))
    ) ) )
  (i (lambda ( ) (quote ((c c c) (m m m) ( ) ( ) left))))
  (f (lambda ( ) (quote ((c c c) (m m m) ( ) ( ) right))))
  (fullmoveset
    (lambda ( )
      (quote (((c c) ( )) ((c) (m)) (( ) (m m)) ((c) ( )) (( ) (m))))
    ) )
  (feasible
    (lambda (move state)
      (cond ((smaller (car state) (car move)) nil)
            ((smaller (cadr state) (cadr move)) nil)
            (t t)
      )
    ) )
  (admissible
    (lambda (state)
      (cond ((null (cadr state)) t)
            ((null (caddr state)) t)
            (t (ofequallength (car state) (cadr state)))
      )
    ) )
  (result
    (lambda (move state)

```

```

160205 -      1

      (list (inc (caddr state) (car move))
            (inc (caddr state) (cadr move))
            (dec (car state) (car move))
            (dec (cadr state) (cadr move))
            (other (caddr state)))
) ) )

(other
  (lambda (riverside)
    (cond ((equal riverside (quote left)) (quote right))
          (t (quote left)))
  ) ) )

(list
  (lambda (a b c d e)
    (cons a (cons b (cons c (cons d (cons e nil))))))
) )

(smaller
  (lambda (x y)
    (cond ((null y) nil)
          ((null x) t)
          (t (smaller (cdr x) (cdr y))))
  ) ) )

(inc
  (lambda (x y)
    (cond ((null y) x)
          (t (inc (cons (car y) x) (cdr y))))
  ) ) )

(dec
  (lambda (x y)
    (cond ((null y) x)
          (t (dec (cdr x) (cdr y))))
  ) ) )

(ofequallength
  (lambda (x y)
    (cond ((null x) (null y))
          ((null y) nil)
          (t (ofequallength (cdr x) (cdr y))))
  ) ) )

(null (lambda (x) (equal x nil)))

(append
  (lambda (x y)
    (cond ((null x) y)
          (t (cons (car x) (append (cdr x) y))))
  ) ) )

(not (lambda (x) (equal x nil)))

(member
  (lambda (x y)
    (cond ((null y) nil)
          ((equal x (car y)) t)
          (t (member x (cdr y))))
  ) ) )

```



```

160205 -      1

(cadr (lambda (x) (car (cdr x))))

(caddr (lambda (x) (car (cdr (cdr x)))))

(cadddr (lambda (x) (car (cdr (cdr (cdr x))))))

(caddddr (lambda (x) (car (cdr (cdr (cdr (cdr x)))))))

))

(crossriver complete try try1 i f fullmovees feasible admissible result other list smaller inc dec ofequallen null append not member cadr caddr
caddr cadddr)
crossriver ( )

garbage collector: 1200

garbage collector: 1031

garbage collector: 1137

garbage collector: 1246

garbage collector: 1020

garbage collector: 901

garbage collector: 1069

(((c c c) (m m m) nil nil right)
 ((c c) nil (c) (m m m) left)
 ((c c) (m m m) (c) nil right)
 ((c c c) nil nil (m m m) left)
 ((c) (m m m) (c c) nil right)
 ((c c) (m m) (c) (m) left)
 ((c c) (m m) (c) (m) right)
 ((c) (m m m) (c c) nil left)
 ((c c c) nil nil (m m m) right)
 ((c c) (m m m) (c) nil left)
 ((c c) nil (c) (m m m) right)
 ((c c c) (m m m) nil nil left)
)
(((c c c) (m m m) nil nil right)
 ((c) (m) (c c) (m m) left)
 ((c c) (m m m) (c) nil right)
 ((c c c) nil nil (m m m) left)
 ((c) (m m m) (c c) nil right)
 ((c c) (m m) (c) (m) left)
 ((c c) (m m) (c) (m) right)
 ((c) (m m m) (c c) nil left)
 ((c c c) nil nil (m m m) right)
 ((c c) (m m m) (c) nil left)
 ((c c) nil (c) (m m m) right)
 ((c c c) (m m m) nil nil left)
)
(((c c c) (m m m) nil nil right)
 ((c c) nil (c) (m m m) left)
 ((c c) (m m m) (c) nil right)
 ((c c c) nil nil (m m m) left)
 ((c) (m m m) (c c) nil right)

```

```
160205 -      1

((c c) (m m) (c) (m) left)
((c c) (m m) (c) (m) right)
((c) (m m m) (c c) nil left)
((c c c) nil nil (m m m) right)
((c c) (m m m) (c) nil left)
((c) (m) (c c) (m m) right)
((c c c) (m m m) nil nil left)
)
((c c c) (m m m) nil nil right)
((c) (m) (c c) (m m) left)
((c c) (m m m) (c) nil right)
((c c c) nil nil (m m m) left)
((c) (m m m) (c c) nil right)
((c c) (m m) (c) (m) left)
((c c) (m m) (c) (m) right)
((c) (m m m) (c c) nil left)
((c c c) nil nil (m m m) right)
((c c) (m m m) (c) nil left)
((c) (m) (c c) (m m) right)
((c c c) (m m m) nil nil left)
)
)

+++ error: eof
```

160205 - 1

compiler data

number of instructions executed: 7497415
 compile time (microseconds): 29111407.50
 average instruction time: 3.88
 object--program length: 5128

execution data

number of instructions executed: 135622946
 execution time (microseconds): 636706448.75
 average instruction time: 4.69

profile

linenumber	count	time	%
8	14	48.75	0.0
11	12	48.75	0.0
41	158	716.25	0.0
42	66	291.25	0.0
43	105	466.25	0.0
44	72	361.25	0.0
50	684	2898.75	0.0
51	14	60.00	0.0
59	42	148.75	0.0
60	756119	4109061.25	0.6
61	5474	23301.25	0.0
62	434	1741.25	0.0
63	45066	200077.50	0.0
64	238119	1383051.25	0.2
65	669724	3311585.00	0.5
66	35	131.25	0.0
67	70	297.50	0.0
68	623521	2699332.50	0.4
69	4079	22691.25	0.0
74	120518	544088.75	0.1
75	242412	1234542.50	0.2
76	262236	1357361.25	0.2
77	722748	3022110.00	0.5
84	2154	8526.25	0.0
85	120731	552477.50	0.1
86	222111	1088785.00	0.2
87	996457	4596528.75	0.7
88	897	2990.00	0.0
89	67669	314177.50	0.0
91	2280	10275.00	0.0
92	300	1200.00	0.0
97	2872	16603.75	0.0
102	18600	93750.00	0.0
103	40886	186115.00	0.0
104	124240	561250.00	0.1
105	19292	89570.00	0.0
107	154	647.50	0.0
108	70	280.00	0.0
109	4800	27750.00	0.0
113	453008	2741202.50	0.4
116	260384	1292507.50	0.2
119	535648	2887477.50	0.5
122	14112	73920.00	0.0
125	767322	3836610.00	0.6
127	2813514	13108417.50	2.1
131	826914	4134570.00	0.6
133	3032018	14126447.50	2.2
139	324005	1215088.75	0.2
140	1618975	6718746.25	1.1
141	3950299	19184853.75	3.0
145	2100488	9604205.00	1.5
148	476388	2018130.00	0.3
149	1652364	6893145.00	1.1
153	112645	469748.75	0.1
154	100920	421225.00	0.1
161	13348041	93295421.25	14.7
162	10057008	58837205.00	9.2
163	1351252	6997555.00	1.1
164	216635	910452.50	0.1

165	37320	160942.50	0.0
166	2799	9330.00	0.0
168	1428	5652.50	0.0
170	186980	794665.00	0.1
172	692010	4613400.00	0.7
180	867191	3697818.75	0.6
181	40265	178305.00	0.0
182	21956	96681.25	0.0
184	21002	95106.25	0.0
185	10589	45745.00	0.0
186	21716	89556.25	0.0
187	18889	77670.00	0.0
188	6600	28050.00	0.0
189	6600	28050.00	0.0
190	85500	607962.50	0.1
191	6649	28242.50	0.0
192	29567	137967.50	0.0
202	881816	3690371.25	0.6
205	67720	275047.50	0.0
206	36702	149102.50	0.0
223	27643	121162.50	0.0
224	199963	985903.75	0.2
225	2154	8526.25	0.0
226	26878	117722.50	0.0
227	77818	347760.00	0.1
228	5466	20227.50	0.0
229	15437	68658.75	0.0
234	19102	77602.50	0.0
235	7748	29835.00	0.0
237	18410	79733.75	0.0
239	1285	4818.75	0.0
240	15934	64892.50	0.0
241	1285	4818.75	0.0
242	481	1966.25	0.0
243	26352	109190.00	0.0
244	31720	133895.00	0.0
245	23180	99430.00	0.0
246	1220	4575.00	0.0
253	7000	27120.00	0.0
254	43848	184440.00	0.0
255	33060	141810.00	0.0
257	7808	31720.00	0.0
258	1464	6405.00	0.0
260	1028	5782.50	0.0
280	113424	484817.50	0.1
281	15720	66810.00	0.0
282	54055	219471.25	0.0
283	51800	210000.00	0.0
284	16025	68368.75	0.0
285	10565	44901.25	0.0
287	54938	250918.75	0.0
289	435	1718.75	0.0
298	13050	51922.50	0.0
299	3393	13507.50	0.0
300	22394	88735.00	0.0
301	10614	42117.50	0.0
302	13833	54881.25	0.0
303	3654	14568.75	0.0
304	2610	10417.50	0.0
305	3915	15521.25	0.0
306	10788	42730.00	0.0
309	16791	66816.25	0.0
310	19140	75707.50	0.0
311	11136	44345.00	0.0
312	5742	22788.75	0.0
313	2436	9655.00	0.0
314	1479	5881.25	0.0
315	10875	43235.00	0.0
316	6177	24530.00	0.0
317	14355	57037.50	0.0
318	3480	13840.00	0.0
319	2001	7932.50	0.0
320	348	1386.25	0.0
321	2697	10697.50	0.0
322	2523	10032.50	0.0
326	31507	135030.00	0.0
330	43560	182256.25	0.0

342	43560	182256.25	0.0
343	44084	177160.00	0.0
344	175274	781645.00	0.1
345	183862	856426.25	0.1
348	13896	58141.25	0.0
349	1351	5307.50	0.0
350	36284	147646.25	0.0
351	1351	5307.50	0.0
352	2346	10488.75	0.0
353	147680	684947.50	0.1
371	3865	17392.50	0.0
372	85437	407838.75	0.1
373	59160	239975.00	0.0
374	274572	1260632.50	0.2
375	258680	1058500.00	0.2
376	254860	1176562.50	0.2
377	257712	1200530.00	0.2
378	36	210.00	0.0
379	49880	202275.00	0.0
385	111312	465732.50	0.1
386	49794	208922.50	0.0
387	87000	365400.00	0.1
402	42436	170465.00	0.0
403	79104	437235.00	0.1
404	397292	1713160.00	0.3
409	48443	201685.00	0.0
410	13896	58141.25	0.0
411	8440	35420.00	0.0
412	130416	545820.00	0.1
413	87828	366610.00	0.1
416	55	231.25	0.0
417	637	2732.50	0.0
418	3848	16250.00	0.0
419	3796	16120.00	0.0
420	208	910.00	0.0
421	3696	15655.00	0.0
422	28	138.75	0.0
423	208	910.00	0.0
424	282	1233.75	0.0
425	6106	26712.50	0.0
427	30	125.00	0.0
428	9100	37895.00	0.0
429	208	910.00	0.0
430	3792	16140.00	0.0
431	28	110.00	0.0
432	7592	32110.00	0.0
434	520	2248.75	0.0
436	49408	206027.50	0.0
455	43560	182256.25	0.0
457	71576	298131.25	0.0
459	33196	138236.25	0.0
465	150	531.25	0.0
466	6650	27437.50	0.0
476	2040572	8258462.50	1.3
477	83630	376335.00	0.1
478	1254450	5435950.00	0.9
482	1832790	7525340.00	1.2
483	3308032	13816600.00	2.2
484	1160992	4910360.00	0.8
489	1832790	7525340.00	1.2
490	2433312	10138800.00	1.6
491	588448	2485000.00	0.4
497	2285064	9560771.25	1.5
498	5731672	23863957.50	3.7
499	35748	183705.00	0.0
501	1233802	5210312.50	0.8
502	2100798	8628277.50	1.4
503	328944	1380390.00	0.2
504	1831662	7522897.50	1.2
505	36735	156123.75	0.0
506	1458600	5916625.00	0.9
507	4418700	18697250.00	2.9
508	3203200	13245375.00	2.1
516	1435320	6005418.75	0.9
517	28602	107257.50	0.0
518	2181074	9010300.00	1.4
520	1940778	8033017.50	1.3

522	1352393	5580240.00	0.9
524	1040508	4273515.00	0.7
525	955136	3998320.00	0.6
526	9710	44908.75	0.0
527	14877	72318.75	0.0
530	709884	2915595.00	0.5
535	709884	2915595.00	0.5
540	709884	2915595.00	0.5
546	709884	2915595.00	0.5
552	709884	2915595.00	0.5
555	709884	2915595.00	0.5
558	709884	2915595.00	0.5
561	709884	2915595.00	0.5
564	709884	2915595.00	0.5
567	709884	2915595.00	0.5
569	422550	1774710.00	0.3
571	1431036	5972040.00	0.9
572	1752174	7359412.50	1.2
573	1256382	5288917.50	0.8
579	300366	1268437.50	0.2
580	1152756	4789620.00	0.8
581	144225	605745.00	0.1
582	328944	1401750.00	0.2
587	4310208	17702640.00	2.8
588	114400	589875.00	0.1
589	4499208	18589095.00	2.9
590	6967800	29712690.00	4.7
591	119448	572355.00	0.1
592	2647764	10924515.00	1.7
601	640	2600.00	0.0
602	920	4175.00	0.0
603	140	600.00	0.0
607	10	40.00	0.0
608	15	55.00	0.0
609	140	552.50	0.0
610	140	552.50	0.0
611	140	552.50	0.0
612	140	552.50	0.0
613	140	552.50	0.0
614	140	552.50	0.0
615	140	552.50	0.0
616	140	552.50	0.0
617	140	552.50	0.0
618	140	552.50	0.0
620	6	23.75	0.0
622	14	48.75	0.0
623	87974	452356.25	0.1
624	34	150.00	0.0
643	1516	6477.50	0.0
644	74	320.00	0.0
645	32	130.00	0.0
646	28	96.25	0.0
647	678	2793.75	0.0
648	296	1292.50	0.0
649	76	327.50	0.0
650	248	1005.00	0.0
651	321	1332.50	0.0
652	3252	13130.00	0.0
653	5064	20820.00	0.0
654	4320	17520.00	0.0
655	6792	28380.00	0.0
656	1728	7080.00	0.0
657	4518	18692.50	0.0
659	252	1047.50	0.0
661	102	413.75	0.0
662	98	405.00	0.0
664	144	590.00	0.0
667	103	471.25	0.0

Appendix B

User–manual extracts

B.1 Extract of the X1 ALGOL user manual

Introduction

...

The text of an Algol program, drafted according the Algol 60 rules (see [1]) with due observance of the rules that apply to the MC Algol system, is typed on a so-called "Flexowriter". This is a typewriter that can record all that is typed on seven-track paper tape too. Such a tape, produced on a Flexowriter and recording the text of an Algol 60 program can be elaborated by the X1 without more ado. This elaboration takes place in two phases.

In the first phase the compilation program, the so-called MC Algol compiler, is activated. This program reads the seven-track tape with the Algol text and compiles this Algol description in a computer prescription that is more adapted to the requirements that are posed on an efficient execution on the X1. The result of this compilation work, the so-called "object program", is punched meanwhile. In this way the compiler produces from an Algol program an equivalent object program. After this the seven-track tape with the Algol text has finished its job and the second phase can start.

In the second phase the computation is actually executed. Since the object program is formulated as a series of standard operations that are collected in a so-called "complex",

¹The X1 ALGOL manual was written in Dutch and is part of the 'MCP' documentation 'P100'. Translation by the author.

in the second phase first the complex is placed in the X1 store. Next the object-program tape produced by the compiler is read by a special input program and recorded in the X1. Then the machine is ready to execute the computation demanded by the program.

...

The complexes ALD and ALS

The standard operations necessary for the object program are, as remarked above, collected in a complex. To the standard operations of the complex belong:

- 1) administrative operations
- 2) the elementary arithmetic operations
- 3) the standard functions:
abs, sign, sqrt, sin, cos, ln, exp, entier.

For the meaning of these functions see the Algol 60 report [1] sections 3.2.4 and 3.2.5. Contrary to what is posed there, in the MC Algol system the value of "abs" always has the same type as its argument. (The standard function "arctan" is not included in the complex but is available as MCP (see in the sequel)).

- 4) the input and output procedures:
read, stop, print, TAB, NLCR, SPACE.

The meaning of these procedures is explained below.

At this moment there exist two complexes, namely the complex ALD, in which the **real** arithmetic is carried out in double length, i.e. in 12 to 15 decimals, and the complex ALS, in which the **real** arithmetic occurs in single length, i.e. in over 7 decimals².

For the arithmetic in both complexes the following holds.

An **integer** variable occupies one single memory location. Consequently its absolute value is less than 2^{26} (= 67 108 864). An anonymous intermediate result that according to Algol 60 has to be of type **integer** but exceeds the integer capacity is transformed automatically to **real** representation.

The **real** variables occupy two memory locations. Anonymous intermediate results of type **real** occupy four memory locations (in the accumulator stack, see various publications of E.W. Dijkstra). Variables and anonymous intermediate results of type **real** are represented in over 7 decimals relative precision in the system ALS. In the system ALD anonymous intermediate results of type **real** are represented in 15 decimals, **real** variables

²The complex ALS was seldom used and not maintained from 1963 onwards.

in 12 decimals relative precision.

The arithmetic operations deliver always unambiguous results. Relations between two numbers of equal type are, moreover, exact. (For an expression E without side effects $E = E$ always holds.) If the operands in a relation are of different type then the **integer** operand is first transformed to **real** representation and next the intended relation is established. The transformation from **integer** to **real** is exact except for **integer** 0. This is transformed to the smallest positive **real** number that can be represented as anonymous number in the accumulator. The same result is formed if sum or difference of two **real** numbers had to be exactly 0. (For an expression E without side effects $E - E = 0$ holds.) Sum, difference or product of two numbers of different type is formed in this manner: first the **integer** operand is converted into **real** representation and thereupon the operation concerned is executed. Possible **integer** operands of the operator "/" are converted into **real** representation, whereafter the quotient of two **real** numbers is formed.

For the operation "==" the following holds: According to the Algol 60 report [1] Section 4.2.4. a transfer function is inserted if necessary. Naturally an assignment of an **integer** result to an **integer** variable occurs exactly. In the case of an assignment of a **real** result to a **real** variable, in the complex ALD the result, represented in the accumulator in 15 decimals relative precision, is rounded to 12 decimals. In the complex ALS no rounding takes place and the assignment is exact, except in the following case. In both complexes a result that in absolute value is too large or too close to 0 is brought within the more restricted range of variables of type **real**. If this form of "overflow" or "underflow" occurs the absolute value of the variable becomes maximal (ca. 10^{616}) or minimal (ca. 10^{-616}) with the sign of the result to be assigned. (As a consequence the relation "a = 0" can never have the value **true** for variable a of type **real**).

The MCPs

The procedures just mentioned in (3) and (4) are available to the programmer without declaration. Also the so-called MCPs have this pleasant property. These were added to the system afterwards and are published in this report with serial letter "AP" followed by a number starting by 100.

The MCP's are not included in the complex but collected in an extension thereof, a so-called "mechanical library". The special reading program of the second phase mentioned in the foregoing reads, immediately after the tape containing the object program, the mechanical library selectively, i.e. only those MCPs are included in the store that are necessary for the object program. This makes it desirable to signal the use of MCPs in a **comment** at the begin of the text of the Algol program.

...

Input and output devices of the X1

In connexion with the execution of Algol programs on the X1 the following input and output devices are of interest:

1) **paper-tape reader**

The paper-tape reader can read both seven-track tapes and five-track ones. During the execution of an Algol program numbers can be read from paper tape by means of the function designator "read". Seven-track tapes should be punched according to the punch code of the MC Flexowriter. In this code each symbol contains a parity bit that is checked at reading. The five-track punch code has no parity bit; thus a programmed check of the input is then desirable. The punch conventions for "read" are discussed below in full detail.

2) **type writer**

For typing of numbers and texts several procedures are available both in the complex and in the MCP library.

3) **seven-track tape punch**

For punching of numbers and texts several MCPs are written. They produce paper tapes punched according to the punch code of the MC Flexowriter. These tapes can not only be typed on the MC Flexowriter, but also be read as input tape by input function "read".

...

Input and output procedures of the complexes

...

Special properties of the MC Algol system

Here the points are enumerated by which the MC Algol system deviates from the Algol 60 report.

...

- 3) Only the first nine symbols of identifiers matter.
...
- 5) Algol 60 allows function procedures to be called as independent statement, besides their use in expressions. In that case the function value is of no interest and is left out of consideration. In the MC Algol system, however, the standard procedures mentioned in sections 3.2.4 and 3.2.5 of [1] and moreover functions "read" and "XEEN" may not be called as independent statement.
- 6) The value of standard function "abs" is of the same type as its argument. Standard function "entier" may have an argument of type **integer**. Functions "sqrt" and "ln" operate on the absolute value of their argument.
- 7) The primaries of an expression are elaborated in the order from left to right.
- 8) Labels starting by a digit are prohibited.
...
- 12) A for clause may not be followed by a conditional statement. In other words: **do if** is prohibited.
...
- 17) Declarations of a block and specifications in a procedure declaration must be given in the following order:
 - 1 scalars (<type> or **own** <type>) and strings
 - 2 arrays
 - 3 destinations (**label** or **switch**)
 - 4 procedures
- 18) Procedures containing array declarations marked by the symbol "**own**" do not function in the official way when used recursively.
- 19) Index bounds in array declarations of the outermost block or in array declarations preceded by "**own**" may only be integer numbers.
...
- 21) The MC Algol system does not discriminate between "**real**" and "**integer**" as first symbol of a function procedure: in each call the type of the result is determined by the arithmetic that is carried out in the procedure body this time.
- 22) Each formal parameter must be specified.
...

- 24) A formal parameter in the **value** list may not be specified as **label** or <type> **procedure**.
 ...
- 26) An array in the value list may have at most five indices.

How to run ALGOL programs

From May 11th 1964 the MC uses officially an ALGOL system³ satisfying the following specifications.

1. Instructions for compiling programs

- a) read the Load-and-go compiler pressing H, 2, 1.
 b) read the ALGOL text pressing H, 2, 0.
 Stop 3-7 means: compilation completed.
 If d1 of the console word equals 0 then label-, switch-, and procedure identifiers are typed⁴ together with their address relative to the start address of the program in the number system with base 32.
 For other stops during compilation see the list of stops, present on the X1 console.

2. Instructions for program execution

autostarts H,0: run the program using the fast paper-tape punch
 autostarts H,G,0: run the program using the slow paper-tape punch.

3. Punch conventions for input data to be read by procedure "read"

...

4. Check on reading and writing outside array bounds

Programs **not** using MCP "INPROD" have only limited checks on reading and writing outside array bounds. After stop 1-23, occurring when an error is found, program execution cannot be continued.

³The "Load-and-go" ALGOL system.

⁴On the IBM typewriter of the system.

B.2 Extract of the X8 ALGOL user manual

1. Requirements to ALGOL 60 programs

The ALGOL system for the X8 accepts programs written in ALGOL 60 as defined in the Revised Report on the Algorithmic Language ALGOL 60, with the provisional restrictions that in the program

- 1.1. no **own** <type> **arrays** are declared,
- 1.2. no integer labels occur with an integer value > 6710883 ,
- 1.3. the number of declared local variables and local labels does not exceed certain, very ample limits,
- 1.4. the number of entries in a switch declaration does not exceed certain, very ample limits.

A number of (function) procedures can be used without needing a declaration in the program. In addition to the elementary functions that are incorporated in Section 3.2.4. of the Revised Report a.o. all input and output procedures belong to them. For a complete enumeration see Section 7.

In addition to the basic symbols of ALGOL the symbols accent, apostrophe, and question mark (', ", and ?), as well as arbitrary combinations or symbols that are underlined or crossed by a bar may occur in programs, but exclusively within comments (after **comment** and **end**) and within strings.

Formal parameters of procedures that do not occur in the **value** part need not to be specified. Usually it is advantageous to specify parameters as much as possible, since the additional information provided by the specification is also taken into account in the formal/actual parameter correspondence checks and moreover can benefit execution efficiency. It is, however, possible to write procedures that derive their special meaning from the absence of one or more specifications.

2. Some interpretations of ALGOL 60

...

⁵Taken from "The MC ALGOL 60 system for the X8. Provisional programmer's manual". Version oktober 1966 (in Dutch), c.f. [10].

2.4. A formal parameter occurring in the **value** part may be specified as **label**. The corresponding actual parameter has to be a designational expression; it is evaluated at procedure entry, just like other parameters from the **value** part.

...

2.6. At the elaboration of expressions the operands are evaluated in the order of the ALGOL text, from left to right. At procedure entry the formal parameters from the **value** part are evaluated in the order of the formal-parameter list; first, however, the simple **value** parameters are dealt with and only thereafter the **value arrays**.

3. Punch conventions for ALGOL programs

...

4. The processing of ALGOL programs

4.1. ...

4.2. Hereafter the processing takes place as follows:

4.2.1. The X8 reads the program and checks it against the syntax in several scans. During the reading the text is printed on the line printer. Each line is preceded by its line number followed by several space symbols. If need the printing of a line is interrupted for the printing of an error message of an error, already detected in the reading phase. For syntax errors detected in subsequent scans an error message is printed after the program text. The representation of the ALGOL symbols on the line printer is treated in table III. Each message of a syntax error has the following form:

er <error number> <line number> <last symbol read>
<value of the last number read> <first 8 characters of the last identifier read>.

The interpretation of error numbers can be found in table I, the decoding of the last symbol read by means of table II. The line number refers to the line of the ALGOL text in which the error is detected. Although the error message gives only 8 characters of the last identifier read, all letters and digits of an identifier do matter for the X8.

Some errors in the ALGOL text can lead to other error messages than would be natural. This is closely bound up with the way in which the syntax analysis of a text that, due to an error, is more or less uninterpretable, is continued.

Each page on the line printer starts with a heading that mentions the date of processing

and a serial number attached by the system to this processing. For text and error messages 60 lines per page are available.

4.2.2. Only if no errors are found during the aforesaid syntax analysis the program is executed. Output on the line printer starts on a new page, output on the paper-tape punch is preceded by a standard prelude and concluded by a standard postlude.

...

4.2.3. Upon detection of an unallowed situation in the execution phase of a program execution is stopped immediately; as last activity an error message is given on the line printer and, if appropriate, a standard postlude is punched on the paper-tape punch. The usual form of an error message during execution is:

er <error number> <line number> <value of last decimal number read>.

For the interpretation of the error numbers we refer again to table I. The line number refers to the line of the ALGOL text where the last statement or the last array declaration starts that was taken into execution but was not yet completed.

4.2.4. ...

4.2.5. Execution of a program is ended:

- a) by "passing" the last **end**,
- b) by calling library procedure EXIT,
- c) by detection of a run-time error,
- d) by operator intervention.

In this last case a pseudo error message is given with error number 999.

...

4.2.6. ...

5. The arithmetic of the ALGOL system

The arithmetic of the X8 ALGOL system corresponds to the floating-point arithmetic of the X8.

5.1. Variables of type **integer** occupy one single memory word. Consequently the value range of integer variables is restricted and runs from $-67\,108\,863$ to $+67\,108\,863$ and no value in absolute value greater than $67\,108\,863$ (after rounding, if not integral) may be assigned to an integer variable during execution of the program. This applies also to implicit assignments to integers (c.f. Revised Report 3.1.4.2., 4.7.3.1., 5.2.4.1., and 5.4.4.).

An exception is the **integer procedure** entier whose value range is not restricted in the above sense.

5.2. Variables of type **real** occupy two memory words and are represented in floating-point representation with a mantissa of 40 binary positions (40 bits) plus sign bit and a binary exponent of 11 bits plus sign bit. In general their values represent non-integral numbers with a relative precision of about 12 decimals. Integral numbers that have an absolute value less than 1 099 511 627 776 can be represented exactly by real variables.

5.3. The elaboration of expressions takes place always in floating-point arithmetic. The arithmetic operations $+$, $-$, \times , and $/$ produce the best possible result that is representable in aforesaid representation, with preservation of the monotony (this implies, e.g., that if $a < b$ and $c \geq 0$, then $a \times c \leq b \times c$).

If the operands to the operations $+$, $-$, or \times are integral numbers with absolute values less than 1 099 511 627 776 and the result of this operation also does not exceed this limit in absolute value, then this result is exact.

The arithmetic operation \uparrow delivers a result with a relative precision of about 12 decimals. If the exponent is an integral number whose absolute value is less than 30 the result is constructed by repeated multiplication.

5.4. The absolute value of expressions and variables of type **real** ranges from about 10^{-616} to about 10^{628} or is exactly 0. If in an arithmetic operation a result is formed that is in absolute value greater than about 10^{628} , then the maximal representable result is formed with the correct sign. Addition or subtraction produces a result 0 only if the absolute values of both operands are equal "bit by bit". Multiplication produces a result 0 only if at least one of the operands equals 0. In all other cases the result of an arithmetic operation has an absolute value of at least about 10^{-616} .

Division by 0 produces, if the denominator differs from 0, the in absolute value greatest representable value. The value $0/0$ can be different from case to case.

5.5. The value of relation $a = b$ is **true** only if the two operands a and b are equal "bit by bit". Relations $a > b$ and $a < b$ produce surely **false** if the two operands are equal "bit by bit".

6. Memory occupation

6.1. For ALGOL programs and their working space (variables, arrays, and block administration as carried out by the ALGOL system) provisionally about 18 000 memory places ("words") are available.

A variable of type **real** occupies two words of memory, variables of type **integer** and **Boolean** one word.

real arrays take two words per element, **integer arrays** one word per element, **Boolean arrays** one word per 27 elements.

6.2. If during execution of a program the available memory space is exhausted, execution is disrupted with an error message with error number 609.

...

7. Library procedures

The preliminary library of the X8 ALGOL system contains the following groups of procedures that need not to be declared in the program:

7.1. Elementary functions

7.1.1. **real procedure** abs (x); **value** x; **real** x;
abs:= **if** x \geq 0 **then** x **else** -x;

7.1.2. **integer procedure** sign (x); **value** x; **real** x;
sign:= **if** x = 0 **then** 0 **else if** x > 0 **then** 1 **else** -1;

7.1.3. **real procedure** sqrt(x); **value** x; **real** x;
If the square root of x is exactly representable in the floating-point arithmetic of the X8, the exact result is delivered. In particular sqrt (i \times i) = i holds for i = 0 (1) 1 048 575. For x < 0 a result 0 is given to sqrt (x).

7.1.4. **real procedure** sin (x); **value** x; **real** x;
The conditions sin (0) = 0 and abs (sin (x)) \leq 1 are satisfied.

7.1.5. **real procedure** cos (x); **value** x; **real** x;
The conditions cos (0) = 1 and abs (sin (x)) \leq 1 are satisfied.

...

7.2. Input procedures

As input device is available a paper-tape reader with a speed of 1000 punchings/second for 5-, 7-, or 8-track paper tape. There are three input procedures for the tape reader:

...

7.3. Output procedures

Provisionally two output devices are available:

- a) a line printer, with a speed of 7 to 20 lines/second. Per page 60 lines are available with a line width of 144 positions;
- b) a paper tape punch for 7-track paper tape with a speed of 150 heptads/second.

...

7.4. Procedures that govern the computation

...

7.5. Various procedures

...

8. Punch conventions for the punching of number tapes

...

9. The timing of several operations

Since as a rule the timing of an operation in the ALGOL system for the X8 depends on the specific syntactic construction the following figures are quite global. In particular the evaluation of formal identifiers will sometimes last relatively long. But the figures given here allow a crude estimation of the execution times of certain program parts. All times are given in μsec .

9.1. The diadic arithmetic operations:

a) integer operands:

+ or -	8	or	20
÷	190		

otherwise as for real operands,

b) real operands:

+ or -	13	or	25
×	40	or	52
/	65	or	77
↑ 2	290		
↑ 3.14	1500		

If two values are given the smaller value holds in the case that the second operand is simple (a constant or a simple non-formal identifier).

9.2. The logical operations:

\neg	5
$\wedge, \vee, \equiv, \supset$	21

9.3. Indexing:

a) integer or real array	50 + 85 per index position
b) Boolean array	150 + 85 per index position

9.4. Assignments:

a) to a real	15
b) to an integer	16
c) to a Boolean	14

9.5. For statements:

for i:= 1 step 1 until n do	80 per iteration
---	------------------

9.6. Block entrance and exit: 45

9.7. Array declaration: 150 + 100 per index position

9.8. Procedure entrance and exit:

110 + 70 per formal parameter

9.9. abs	13
sign	13
entier	80
sqrt	340
sin	470
cos	450
arctan	725
ln	580
exp	735

Bibliography

- [1] J.W.Backus et.al. *Report on the Algorithmic Language ALGOL60*.
Regnecentralen, Copenhagen, 1960
- [2] F.J.M. Barning. *X8-simulator, simulatieprogramma voor het uitvoeren van X8-programma's met behulp van de X1*.
Mathematisch Centrum report R 917, 1964.
- [3] F.J.M. Barning. *Standaardfuncties in X8-Assembler-code ELAN*.
Mathematisch Centrum report R 1042.
- [4] P.A. Businger. *Eigenvalues of a real symmetric matrix by the QR method*.
Algorithm 253, CACM **8** (1965) 217.
- [5] E.W. Dijkstra. *Communication with an automatic computer*.
PhD Thesis University of Amsterdam, 1959.
- [6] E.W. Dijkstra. *The structure of the 'THE' multiprogramming system*.
Comm. ACM **11** (1968), 341 – 346.
- [7] A.A. Grau. *On a floating-point representation for use with algorithmic languages*.
Comm. ACM **5** (1962) 160.
- [8] F.E.J. Kruseman Aretz. *The Dijkstra-Zonneveld ALGOL 60 compiler for the Electrologica X1*.
CWI report SEN-N0301, 2003.
- [9] F.E.J. Kruseman Aretz. *ALGOL 60 translation for everybody*.
Elektronische Datenverarbeitung **6** (1964), 233–244.

- [10] F.E.J. Kruseman Aretz. *Het MC-ALGOL 60-systeem voor de X8. Voorlopige programmeurshandleiding.*
Mathematisch Centrum report MR 81, 1965.
- [11] F.E.J. Kruseman Aretz and B.J. Mailloux. *The ELAN source text of the MC ALGOL 60 system for the EL X8.*
Mathematisch Centrum report MR 84, 1966.
- [12] F.E.J. Kruseman Aretz, P.J.W. ten Hagen, and H.L Oudshoorn. *An ALGOL 60 compiler in ALGOL 60.*
Mathematical Centre Tracts 48, Amsterdam 1973.
- [13] F.E.J. Kruseman Aretz. *On the bookkeeping of source-text line numbers during the execution phase of ALGOL 60 programs.*
in: J.W. de Bakker e.a.. *MC-25 Informatica Symposium.*
Mathematical Centre Tract 37, 1971.
- [14] F.E.J. Kruseman Aretz. *On deriving a LISP program from its specification.*
Science of Computer Programming **10** (1988) 19–32.
- [15] R.N. Kubik. *HAVIE INTEGRATOR.*
Algorithm 257, CACM **8** (1965) 381.
- [16] P.J.J.van de Laarschot and J. Nederkoorn. *User's direction of the second MC ALGOL 60 translator.*
Mathematisch Centrum report MR 64, 1963.
- [17] P.J.J. van de Laarschot and J. Nederkoorn. *Text of the second ALGOL 60 translator for the X1.*
Mathematisch centrum report MR 61, 1963.
- [18] P.Naur (ed.) *Revised Report on the Algorithmic Language ALGOL60.*
Regnecentralen, Copenhagen, 1962
- [19] J.A. Zonneveld. *Automatic Numerical Integration.*
PhD Thesis University of Amsterdam, 1964.