stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 58/76 DECEMBER

H.J. BOOM & E. DE JONG

A CRITICAL COMPARISON OF SEVERAL IMPLEMENTATIONS OF PROGRAMMING LANGUAGES

Preprint

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

AMS(MOS) subject classification scheme (1970): 68A30

ACM-Computing Reviews-categories: 4.6, 4.22, 4.12

A critical comparison of several implementations of programming languages*)

by

H.J. Boom & E. de Jong

ABSTRACT

The implementations of Algol 60, Fortran, Pascal, and Algol 68 provided on the CDC Cyber 73 of the Stichting Academisch Rekencentrum Amsterdam are compared on both qualitative and quantitative aspects.

KEYWORDS AND PHRASES: *Algol 60, Fortran, Pascal, Algol 68, programming language implementation, CDC Cyber 73*

*) This report will be submitted for publication elsewhere

0. INTRODUCTION

Around the end of 1974, it was decided to carry out a comparison of some of the programming language implementations available on the CDC Cyber 73 of the Stichting Academisch Rekencentrum Amsterdam (henceforth known as SARA). The intention was to provide some guidance to new users of this system. In order to limit the scope of this study, four languages were selected for which there was general interest within the Mathematical Centre (MC). These were Algol 60, Algol 68, Fortran, and Pascal.

It soon became clear that running computer programs and measuring execution times and storage consumption would tell less than half of the story. For most programming projects, qualitative aspects of the language are far more important than quantitative ones. The varying facilities available in different languages strongly affect their suitability for different problems. Indeed, in recent years it has become generally known that assembly languages may not be the best tools to use on large systems programming projects, even if efficiency of execution is the most important criterion.The code generated by a good optimizing compiler can be better than that produced by a good assembly language programmer, if the program is large.

The scope of the study has therefore been extended to include various qualitative aspects.

Finally, we must mention that we have found this analysis to be far more difficult and time-consuming than we had originally expected. It may well be that errors have crept into this critique, perhaps because the systems were themselves being changed or replaced by new versions during the study. If so, we would appreciate hearing of them.

0.1. The language implementation

Fortran is the oldest of the four languages studied. Invented during the 1950's, it was the first attempt to construct an algebraic language. It contained very many ad hoc compromises, but achieved widespread popularity because it was first. This language has affected the architecture of

many computer systems.

Algol 60 was constructed around 1960. Responsibility for it was later taken over by IFIP Working Group 2.1. It was one of the first languages whose structure encouraged the so-called "structured programming", but this was not actually discovered until nearly a decade later.

Algol 68 is, in some ways, a descendant of Algol 60. After long considerations, the Algol Working Group decided it was time to begin work on a new language which would be cleaner and more complete than Algol 60 and which would not perpetuate the mistakes of the first attempt. When its first defining Report appeared in 1968, it was seen to be quite different than Algol 60. Final definition of the language was not complete until its Revised Report appeared in 1975 [3], a very long time later indeed.

When Working Group 2.1 produced the Algol 68 Report in 1968, there was a substantial dissenting minority protesting the publication of Algol 68 at that time. Some of them insisted that the Report be made more clear and that an implementation be ready before the Report could be acceptable. This minority resigned from the Working Group when the original Algol 68 Report was published, and hindsight now shows that they may have been right in their severe criticism of the language as presented in 1968. One of them, Niklaus Wirth, produced another language, Pascal, shortly thereafter, perhaps as a form of constructive criticism. By carefully limiting the scope of the language, he was able to define a clean, straightforward, and efficiently implementable language with some severe deficiencies. It was implemented and made available rather quickly on the CDC 6600. In contrast, Algol 68, a much richer language, took another seven years to reach anything like a comparable state.

But it is the implementation of a language that a programmer uses, and in his eyes the implementation becomes inseparable from the language. Each implementer makes his own impact on the user by various deficiencies and extras. Little distinction will therefore be made in the rest of this paper between the implementations and the languages. The following implementations were studied:

Pascal.

Algol 60: CDC Algol 60 version 3.
CDC Algol 60 version 4.

Algol 68: CDC Algol 68 version 1.0.9.

Fortran: CDC Extended Fortran.

In addition, the Minnesota MNF Fortran compiler appears in the timing measurements, but it is not discussed elsewhere.

1. COMPATIBILITY

It is desirable to be able to export programs to other installations and to import programs from other installations. It is even more pleasant if those other installations can achieve reasonable communication even though the computing machinery they possess differs greatly from that locally available. Such "portability" significantly increases the market for any program one wishes to export, and makes it possible to avoid effort by importing a working program instead of writing one locally.

There are essentially two means of transporting programs. First, it may be possible to have them written in a generally available programming language. Second, it may be possible to have them clearly written in a language of such elegant semantics that it becomes very easy to translate them to one of the locally available languages by hand. Translating an existing program usually involves less work than writing a new one, if the original program is easy to understand.

The first approach seems quite attractive, and one must choose the language. It is important that

(1) the language be standard,

(2) an implementation be locally available, and

(3) the implementation indeed implement the language.

If the language does not have a unique definition with some official status, it is extremely unlikely that implementations on different machines will be even slightly compatible. For example, although nearly every large computer has several implementations of Lisp, they differ sufficiently that it is not practical to use Lisp as a language for portable programming.

If an implementation of the language is not available, it is impossible to write debugged programs for export. Import is still possible if one is willing to convert the program by hand, but it is extremely tedious.

It is desirable that the local implementation implement exactly the standard. If it implements a subset of the standard, importing programs becomes difficult. If it implements a superset of the standard, it becomes impossible to be certain that a locally debugged program for export does not accidentally use a superlanguage feature. Such matters may be extremely subtle. An implementation may define some matter which the language definition leaves undefined, such as whether variables are initialized to any specific value. Initialization could be relied on without any explicit mention of the fact within the program.

If the implementation accepts some standard language features but assigns different semantics to them, it will be extremely difficult either to import or export programs. Compilers will not detect such language deviations in a program (the answers will merely be wrong).

Because of practical difficulties, it will usually be necessary to make some small changes in a program upon transportation even if a conscientious attempt was made to adhere to the standard language. It is then of great importance that the program be readable.

It may in some cases be easier to hand-recode a program written clearly but in a locally unavailable language than to alter a confusing program written in a locally available language with slight deviations.

1.1. Pascal

Pascal is defined by a defining Report [1]. This Report is accompanied by an appendix describing details of the implementation on the CDC 6600. This implementation does appear to conform closely to the Report; it appears that other implementations are likely to do so too. Nonetheless, it is not clear to what extent the Pascal implementation for the Cyber is compatible with implementations on other machines. Other implementations are only now appearing, and reports of experience with them has not yet reached a general audience. The Report leaves ample room for implementers to use machine-dependent criteria such as the size of a machine word to determine a number of details. It would be reasonable if this extended to matters such as the precision of arithmetic, but at a number of points these limitations can be expected to affect program correctness severely. The following list

contains relevant parameters:

- the number of significant characters in an identifier. (Extra ones are legal and ignored. This can be disastrous if one attempts to transport a program and finds that formerly distinct identifiers have become identical, or vice versa.)
- the size and coding of the character set.
- the number of elements permitted in a power set. (The CDC implementation permits 59. This means that a *set of char* is impossible, because 64 characters are recognized in the character set.)
- the number of characters in a value of type *alfa*.

Other implementations will probably find other ways to impose annoying qualitative restrictions by propagating low-level machine-dependencies to the level of the high-level language. Pascal provides high-level concepts, but restricts them so that the programmer has to think in machine terms.

Whether a program violates the above constraints is a matter that can easily be determined at compile-time. There seems to be no reason why the CDC compiler should not compile code for these prohibited cases anyway, perhaps by using more storage for larger objects, without impairing run-time efficiency one whit for the non-user. The CDC compiler has set an example of machine-dependent restrictions which we must hope other implementations will not follow.

1.2. Algol 68

Algol 68 is defined in the Revised Report on the Algorithmic Language ALGOL 68 [3], hereinafter called "the Report", or "the Algol 68 report". This Report is virtually impossible for the uninitiated reader to understand, and may be difficult even for the experienced Algol 68 programmer. This Report is, on the other hand, extremely precise. It even makes explicit at which points the implementer has freedom to make implementation choices.

The CDC implementation has adhered extremely closely to the specifications in the Report. There are a number of unimportant deviations from the Report, and a number of minor language extensions. These are all clearly

mentioned in the CDC Algol 68 reference manual.

The CDC implementation is closer to the spirit and letter of the Revised Report than any other implementation published for any machine that the authors know of. Unfortunately, there are hardly any other complete implementations in existence, and therefore this fact does not at present aid portability.

The most important language deviation is concerned with the opening of input files. It is not possible to do this, under normal conditions, by using the *open* routine. It is clearly the intention of the Report that the *open* routine be used for this. Instead, *establish* must be used to provide the system with a number of characters per line, number of lines per page, etc. This is further discussed in the section on input/output.

1.3. Fortran

Fortran has been standardized by ISO and ANSI. Unfortunately, the definition of Standard Fortran is extremely difficult to read and understand. Even experts in Standard Fortran regularly discover new catches or properties of the language every year. Those interested in an introduction to Standard Fortran are advised to consult the Standard Fortran Programming manual, which contains a reprint of the standard and much useful advice [6,7,8,9,10].

The CDC implementation pretends to make a clear distinction between its standard subset and its nonstandard extensions. The Fortran Extended manual indicates this by shading descriptions of nonstandard features in grey. Unfortunately, when there is a deviation from the standard, only the extended version is described, and not the standard one. For example, CDC Extended Fortran accepts variable names of up to seven characters. In the manual, the "seven" is shaded, but it is nowhere mentioned that the limit in the standard language is six.

The attempt to distinguish between the standard and the implemented language must be praised, even though the omissions can be seriously misleading.

On the other hand, a programmer fully cognizant of the Fortran standard can write Standard Fortran programs and have them accepted by the Fortran Extended compiler with only minimal change (a PROGRAM statement

is necessary at the beginning to describe input and output files), and Standard Fortran programs from elsewhere can be easily imported. A fortran compiler that detects all deviations from the standard, but still processes all Standard Fortran programs correctly, would need extremely complicated run-time checks on use and misuse of common storage, variables initialized in DATA statements and subsequently assigned to, and many other matters. To our knowledge, no such rigorously checking compilers have ever been written for Standard Fortran.

The reason for placing especial emphasis on the standard for Fortran is that virtually every computer in the world has at least one compiler available which will accept a superset of Standard Fortran. This is more true of Fortran than of any other programming language. Writing a program in Standard Fortran, distasteful though it may be, or having it mechanically translated to Standard Fortran, is therefore an effective method of achieving machine-independence. The programmer should be warned, however, that Standard Fortran is probably but a small subset of the language he thinks of as Fortran.

Recently, there has been work on a new Fortran standard [2]. This proposal has not yet been formally accepted, and the new standard has not yet been generally implemented. It is therefore not a useful vehicle for achieving portability.

1.4. Algol 60

Algol 60 was defined in 1962 by The Revised Report on the Algorithmic Language Algol 60 [5]. In recent years IFIP Working Group 2.1, the group which is responsible for Algol, has had second thoughts based on more than a decade of experience with the language, and has approved for publication a document [17] making minor changes to the language and clearing up a number of subtle confusions and ambiguities in its definition. One effect of these changes is that the modified language [18] (which may become known as Algol 60.1) is actually closer to most existing Algol 60 implementations than that of the original Revised Report.

The Revised Report is not an obscure document, but it is written as a

language definition and not as a tutorial. CDC has reprinted it in Chapter 2 of their Algol 60 manual, together with a large number of insertions (in a different type face) describing deviations from the report and giving details about machine-dependent matters. The restrictions are, in general, those made by many implementers, and should not seriously hinder program portability.

Since Algol 60.1 appears likely to supersede Algol 60, we shall here mention the more important incompatibilities between Algol 60.1 and CDC Algol version 4. The deviations from Algol 60 are abundantly clear from the CDC reference manual.

Algol 60.1 provides the following standard procedures:

abs, sign,
sqrt, sin, cos, arctan, ln, exp,
inreal, outreal,
maxreal, minreal, maxint, epsilon, entier, iabs,
fault, stop,
inchar, outchar, ininteger, outinteger
outterminator, outstring, length.

CDC Algol 4 provides those on the first three lines (*abs* to *outreal*), but does not provide the rest (*maxreal* to *length*). However, the features *maxreal, maxint, epsilon, inchar, outchar,* and *length* are provided in other forms.

Algol 60.1 deviates in its definition of the type of integer exponentiation. If the base of the exponentiation is of integral type, and the exponent is an <integer> or a call on the function *iabs,* then the type of the result is integral, otherwise real. Algol 60.1 provides a result of type integral if the base and exponent are both of type integral; use of a negative exponent is then unlawful.

Algol 4 does not permit the entire program to be labelled; Algol 60.1 does.

Algol 60.1 treat a for loop as if it were a block; the scope of any label preceding the loop body is thus limited to the loop, and it is impossible to jump into the loop from outside. It is possible to use the same label inside and outside the loop. Algol 4 follows the older Algol 60 rules on this matter, prohibiting a jump to a label inside a loop but also prohibiting use of the label in the block containing the for loop.

The original Algol 60 report did not provide any input/output facilities; it was felt that machines differed too widely to make standardized input/output feasible. The effect is that each implementer constructed his own input/output system. It is therefore advised that programmers writing portable programs should concentrate the input/output in a small number of small and simple procedures, which can easily be replaced.

Nonetheless, there have been proposals for extended input/output systems, and CDC has implemented that of Knuth et al. [14], with modifications. Chapter 3 of the CDC manual contains the Knuth proposal, with modifications, in the same style as chapter 2.

2. RELIABILITY

It is not sufficient that the programmer, with one finger on the language definition and one on the coding sheet, can write texts which resemble syntactically correct programs. He must also be able to run such a program on a real machine, correct any errors it might contain, and ascertain that it does then perform reliably.

The behaviour of the language and of the implementation has enormous influence on debugging. The implementation itself must reliably conform to specifications, the specifications must be clear, simple, and useful, and the language and implementation must together prevent errors and clearly report those which do occur. We can distinguish a number of specific requirements.

The implementation itself must work, and be fully debugged. If a program fails, the programmer must be able to be certain that the fault lies with the program and not with the implementation. Nonetheless, if there are implementation errors, they must be well published and swiftly repaired.

The language must actively help a programmer to structure his programs. This does not mean that it must straightjacket the programmer into one specific approved style of program construction; it must instead provide primitives that are of use in forming structure, and detecting accidental violations of any structure the programmer himself imposes. The language must, furthermore, refrain from providing the unwary with traps.

{ There are more ways to structure a
program than a man can shake a stick at.

One man's bug is another man's
structure.

- Traditional }

The implementation must then help the programmer to find the errors remaining in the program. It must be possible for the implementation to catch all language violations. It must be highly likely that programmer errors lead to such language violations, preferably ones that are detected at compile time. It must be easy for the programmer to request such thorough checking. When an error is detected, it must be easy for the programmer to find it. The implementation should assist him, providing a reasonable amount of post-mortem information in a readable form. The implementation may not run amok, providing false or misleading messages or forcing the programmer to wade through octal or similar core dumps.

Complete checking has two virtues.

First, it can signal the presence of certain program bugs, to wit, those which cause the program to violate language restrictions. Even if checking were only 98% complete, bug-detection would not be significantly impaired. A bug which fails to be detected by one possible but absent check will likely be caught by another.

Second, it can be used in finding the error. For example, suppose one wishes to know at which point in a program a variable receives an anomalous value. It is an enormous help to know that this cannot happen through the use of an out-of-bounds subscript in an apparently irrelevant assignment. The fact of complete checking can thus be used in logical deductive reasoning to reduce the search domain drastically. This property is completely lost if checking is only 98% complete.

The fact of complete checking, together with a selective and readable post-mortem dump, is often more useful than run-time tracing of jumps, assignments to specific variables, and the like. Complete checking, moreover, does not have to be planned in advance; whereas the more traditional traces must be carefully used in further runs after a bug has been detected.

2.1. Pascal

Until April, 1975, errors were found in the Pascal compiler in use at SARA, and new releases appeared approximately every 1½ to 2 months. The latest release was received in August, 1975, and no errors in it have come to the authors' attention. It thus appears to be of reasonably solid construction. This is perhaps because the current version of the compiler was itself written in Pascal. This makes reasonable clarity of code possible, and makes the compiler itself one of its own test cases. Unfortunately, when one examines the source code of the compiler itself, one finds it written in an unreadable and nearly comment-free style.

Identifiers may contain only 10 significant characters; extra characters may be coded, but are ignored without warning by the compiler. It is thus easy for a programmer to code two apparently different identifiers and have the compiler misinterpret the program by failing to recognise the difference. This can be catastrophic if the two identifiers are of the same type, since the error can then go completely undetected.

Syntactic error recovery is good; it is extremely rare to get two error messages for one single syntax error. However, missing or extra begins or ends can cause the compiler to fail to properly identify identifiers, which can cause much trouble. Nonetheless, the compiler rarely loses all track of the intended syntactic structure, and therefore it is possible to remove syntactic errors in relatively few runs.

Run-time checking is incomplete. There are a number of points where program errors can lead to incomprehensible and undefinable chaos. Two serious problems are variants, and the parameters of parameters.

A Pascal record may have "variants", which means that at various times, different fields may be present in the record. (The record corresponds to the Algol 68 structure, and the variants to united modes.) Unfortunately, there is no built-in check to ensure, when a field of some variant is used, that the variant with that field indeed does reside in the record at that time. This can be used for intentional or unintentional punning. As Niklaus Wirth says [22], assembly language programmers delight in ingeniously misusing features provided with honest intentions to betray the language's very principles. The serious high-level language user can only the lack of

security and the resulting failure to find programming errors easily. We have the following example of a coding trick. This is a program which prints the contents of the first 4000 words of memory. It could just as well have overwritten the first 4000 words of memory, at least until the program gets so far as to overwrite itself:

```
program tt(output);
type rec = record f1 : integer;
       case f3 : boolean of
         true : (f4 : integer);
         false : (f5 : ↑ alfa)
       end;
var a : rec;
   i : integer;
begin write ('0') : for i := 1 to 4000 do
  begin a.f4 := i; write (' ', a.f5↑); if i mod 10 = 0 then
     begin writeln; write (' ') end
  end;
  writeln
end.
```

When writing a procedure which accepts a procedure as a parameter, there is no way to specify the types of the parameter to the parameter, although these are usually known to the programmer. There is therefore no compile-time check on the compatibility of parameter type checking in such cases. There appears to be no run-time check either.

Pascal does provide list processing, but does not provide a garbage collector. This means that storage allocation and freeing must be explicitly coded by the programmer, with the attendant risk of catastrophic error. Storage allocation is done using the procedures *new* for allocation and *dispose* for freeing. If the storage freed by *dispose* is reused by *new*, there is danger that the now reused storage is still pointed to by a pointer left over from its previous use. This can cause interactions between

independent parts of a program that are extremely difficult to diagnose. If the storage is not reused by *new*, there is no sense in using *dispose* at all, and any serious attempt to do list processing will fail when memory becomes full of useless list cells that cannot be reused.

There are indications that the version of Pascal in use at SARA may be a modified version that does reuse the storage. However, since there is no compactifying garbage collector, there may be danger of storage fragmentation if allocated records are of different sizes. This means that freed pieces of memory may be splintered by further allocation, leaving splinters free storage too small for reuse.

There seems to be no secure way of implementing the language defined by the Pascal report on conventional computers without going to prohibitive expense, by providing tag bits on every value for dynamic type checks. Without such a run-time mechanism, Pascal is not type-secure. A garbage collector is therefore not a possibility; programmers will therefore have to make do with an insecure language.

At program termination, Pascal provides a symbolic dump of the run-time stack, including the names of variables. Unfortunately, the elements of arrays and records are not printed, the records allocated by *new* are not printed, and nothing at all sensible can be printed if the above-mentioned insecure use of pointers has seriously damaged the stack.

2.2. Algol 68

With version 1.0.8, the CDC Algol 68 implementation had reached a relatively bug-free state. Until then, it was still under development, it was undergoing continuous changes, and as result it was extremely buggy. Bugs are still found, but rarely, and are usually fixed within a few months. Most bugs appear to reside in the garbage collector or the code generator, and their effects disappear when Algol 68 source code is replaced by different, but functionally identical source code. Re-coding a statement to cause different register assignment or changing object-time field length usually suffices, but the presence of such bugs must still be considered a serious difficulty.

By default, the compiler is in a state in which most language violations are caught at compile or run time. The implementation does not run amok (except as mentioned below). In practice, it appears that most

programmer errors are detected by the compiler at compile-time, usually by the compatibility check on modes. The errors detected in practice at run time are mainly of the "undefined variable" type: the omission or misplacement of initialization. Very few programmer errors indeed survive both the compile-time and the run-time checks. Such bug-resistance must be construed as a significant advantage.

It is possible to get a readable post-mortem dump from Algol 68. It consists of a printout of the active stack, with variable names and their values. The only serious security risk inherent in the implementation is the occasional failure of scope checking in certain situations involving explicit parallel processing. The scopes of procedures which arise within one parallel process can be confused with those which arise during another parallel process. Programs which do not explicitly use parallel processing have nothing to fear from this security risk. Avoidance of parallelism, furthermore, is not a severe restriction; the authors have yet to see an Algol 68 program using parallel processing that was not specifically written in order to illustrate the feature or to test the compiler.

It should be mentioned that the scope checks are looser in the implementation than in the Report, although they are still secure (except in connexion with parallel processing). All variables are placed on the heap instead of on the stack, and their storage is retrieved by garbage collection. No scope check is done upon assignment (this can save much execution time), but instead a scope check is performed upon procedure calls, to determine that the called procedure's necessary environment still exists. In most cases this check can be performed at compile time (but it is not clear whether it is indeed done then).

2.3. Fortran

The Fortran Extended compiler works. It is not clear whether it is bug-free, since the language implemented is more or less a superset of the standard, and it encourages a "try it and see" attitude. Matters which in other languages would be considered bugs are in Fortran treated as "that's the way it is". For example, in free-format output of integers, the compiler uses heuristics to determine whether a word contains a true integer,

or was probably intended as a character string. Its arithmetic deviates from what one would expect from a first or second reading of the manual (for example, multiplication of integers does not work if the product requires more than 48 bits, although the manual clearly says that integers of up to 59 bits are valid; this restriction in multiplication is mentioned hidden away in the third part of the manual). But, if one is willing to experiment, to accept unexpected limitations, and be constantly aware of the limitations of the machine instructions that the Fortran system will probably use to implement Fortran operations, it can be used. The Fortran system must be seen as a machine-dependent medium-level language. This is true of Fortran on many machines, although Fortran systems on other machines often have less glaring machine deficiencies to fail to hide.

Fortran also leaves much to desire in the direction of clear, comprehensible programming. The almost complete absence of what have now come to be known as "structured programming tools" makes it unsuitable for building large, reliable systems.

By inserting special statements in the source program and further specifying the compile-time "D" option, which causes them not to be ignored, various run-time checks, such as array subscripting checks, can be turned on. These statements all begin with "C$" in columns one and two, and will thus be processed as normal comment cards by other Fortran compilers. (The Minnesota Fortran compiler, which is not itself properly discussed in this paper, uses other conventions for these options.) Unfortunately, such checks are performed only where the implementers thought it convenient to do so, and therefore full security (such as is provided by the Watfor and Watfiv Fortran compilers on the IBM 360) is not provided. The most glaring exception to full subscript checking is that subscripts are not checked in input/output statements. Furthermore, as in Algol 3, a check is made only on whether the final array element is within the entire array, not whether each subscript is within its own proper bounds. Unfortunately, it is also difficult to turn run-time checking on with these "C$" statements. The Fortran manual appears to be very free in the placement of these debug statements, saying they may appear interspersed within normal Fortran statements. Unfortunately, this is not quite true, and there are a few places where debugging statements are ignored unless preceded by an extra

"C$ DEBUG" statement. When we attempted to check that run-time subscript checking did indeed occur by writing a short program with a deliberate subscript violation, it took us four runs before we obtained a run-time error message. Our sympathies go out to the programmer who actually tries to debug his program using the Fortran debugging package. He will not have the advantage of knowing the nature and location of the error beforehand.

(Six months later, one of our colleagues pointed out that the errors we had made were explicitly mentioned in the reference manual; however, this was in a different place than we had looked to find out how to use the debug feature. It is a pity that the authors of the manual have not seen fit to describe all the various rules for placement of debug statements in one single place, instead of placing various parts of the specifications differently. Cross-references could even be a help. Even if the documentation were to be improved, we should still regret that the rules for placement of debug statements are so complicated; a debugging feature should contribute to the solution, not to the problem.)

Of course, since main programs and subroutines are (in principle) compiled separately, there is no check on parameter type compatibility. Furthermore, type conversions, that are normally performed automatically in assignment statements, are not provided for actual parameters, since the compiler does not know what the types are that a subroutine expects for its parameters. No run-time checks are provided for this either.

The main use of Fortran seems to be as a low-level language in which it is possible to reach various features of the hardware or of the operating system directly. This is inherently machine-dependent, and the programmer must be aware of the ways that machine characteristics jut out in unexpected places.

2.4. Algol 60

The object code from Algol 3 and 4 is usually correct; however, in the unusual case that it is not correct, compiler bugs are not corrected promptly by CDC. A delay of one to two years is not unusual.

Algol 3 provides a simple option to turn on checking of array bounds, after which all subscripting is checked for all arrays. Unfortunately, it

does not check whether each subscript is within bounds, but only whether the computed effective address lies within the array.

Algol 4 does not provide a simple option to turn on array-bound checking; it must be specified anew for each array by placing a comment in the block with its declaration. This means that checks can easily be forgotten. On the other hand, when checking does occur, Algol 4 checks that each subscript lies within its proper bounds, and not just that the effective address be in the array.

Algol 3 and Algol 4 both have code optimizing facilities. However, the Q option in Algol 3 is a cruel joke on the programmer by the implementer. According to the manual, it will cause incorrect code to be generated:

> "If a call within a for loop changes the value of a variable accessible to both procedure and for loop and that variable is not an actual parameter of the procedure, then subscript expressions in the for loop which depend on the variable will be evaluated incorrectly ..."
> (p. 2-15, Algol 3 Reference Manual).

As if the job of a programmer were not difficult enough, as if bugs were not persistent enough, that we have to have a compiler that introduces more of them!

Algol 4 appears not to have this defect in the optimization.

3. ARITHMETIC

The hardware of the CDC Cyber is notorious for the poor quality of its arithmetic. It provides no immediate warning of overflow, underflow, or serious loss of significance, and instead yields infinite, indefinite, or nonsensical values and allows computation to continue. Such undetected faults can seriously impair the reliability of numerical results.

It must be granted that only a finite subset of all numbers can be represented on a computer. Operations cannot always be performed exactly, since their exact values may not belong to this finite subset. Nonetheless, it is reasonable to require a number of properties to hold on the operations as implemented by the hardware. For example, one might require:

(1) If the exact result of an operation on specific operands is

exactly representable on the machine, then a representation of that exact result must be produced by the machine operation.

(2) If the mathematically exact operation is monotonically increasing (decreasing) over some range, then the implementation shall not be decreasing (increasing) over that range.

(3) If there is no reasonable approximation to the exact result available, an error will be signalled in an effective manner.

Further conditions, and some discussion on their necessity, have been described by Kuki [11]. Such properties are, in fact, more important than that the computed values be "close" to the true values. Many iterative algorithms do not require high precision, but will fail if one of these requirements (such as monotonicity) is not satisfied.

The arithmetic on the CDC Cyber fails even the first of these requirements. What is even more amazing, it fails to satisfy it on integer arithmetic! The machine ostensibly provides 60-bit integers, 59 bits and a sign bit. It uses one's-complement arithmetic; there are therefore two representations for zero, a +0 and a -0. Correct fixed point addition and subtraction operators are provided (except for overflow), but multiplication fails if the product exceeds 2^{48}. No error indication is provided; the answer is instead just wrong. There is no fixed point divide instruction; floating point division must be used instead, followed by truncation to integer. Division thus also fails on integers greater than 2^{48}.

Floating-point addition and subtraction produce an unnormalized result, which can be separately normalized by a normalize instruction. This implies that under some conditions the last (significant) bit of a computed value is irretrievably lost.

The Cyber appears to satisfy the second requirement except when capacity constraints such as the above are exceeded, but does not satisfy the third one properly. Depending on the operations performed, one may get a nonsensical result or a special value "infinity" upon overflow. In some cases, special "indefinite" values can be produced. If the result is nonsensical, computation can merrily continue, combining nonsense to beget more nonsense. If the result is infinite, an error interrupt is not signalled by hardware until an attempt is made to use the infinite value as an operand. Production of infinite or indefinite values is perhaps tolerable, since it is at least possible to see afterward that something has gone wrong, though it may no longer be easy to find out where. But getting nonsensical answers without warning, as happens when a fixed point multiplication goes out of range, is really inexcusable.

For reasonable reliability, a programming language implementation on the Cyber must find ways of compensating for these deficiencies. The results must be correct, not merely rapidly computed. Unfortunately, proper software compensation for these hardware faults is prohibitively expensive. The most that is usually done is to post warnings in manuals as to the limitations of the implemented arithmetic.

The reasons for the various code sequences generated by the Pascal compiler are discussed in [13] by Wirth. The serious user of the CDC 6600 is strongly advised to read this paper, because its "understanding may prevent him from certain pitfalls which are inherent in the use of the CDC 6600".

We have evaluated a number of expressions on the various implementations and had the results printed. Deviations from mathematically exact results may therefore result from inaccuracies in calculation, comparison, or printing.

The results are summarized in the following table. We have used a number of abbreviations:

0	= 0.000 ... 00E0
0'	= 0.000 ... 00E-295
K	= 3.13151306251402E-294
K'	= 3.1315130625140E-294
K"	= 3.131513062514E-294
π	= 1.56575653125702E-294
π'	= 1.5657565312570E-294
1	= 1.000 ... 00E + 0
	= .1000 ... 00E + 1
$1 - \epsilon$	= .999999999999996E + 0
$1 - \epsilon'$	= .99999999999999E + 0
	= 9.9999999999999E - 1
$1 - \epsilon''$	= .9999999999999E + 0
$(1 - \epsilon)^2$	= .999999999999993E + 0
p	= 2^{-976}
q	= 2^{-975}
C	= cos(0)

In general, the number of significant digits printed has been ignored in reporting these results. In the case of Pascal, Algol 60, and Algol 68, the default number provided by the language was used; for Fortran an explicit format was given.

"q" is the smallest power of two which all the systems concerned could distinguish form zero. "p" is the next smaller power of two. As one can see, it is sometimes distinguishable from zero and sometimes not. Worse, properties such as

$p \neq 0$ implies $1 * p \neq 0$

and

$p \neq 0$ implies $2 * p \neq 0$

appear to fail! With one system, we even have

$p \neq 0$, $1 * p = 0$ but $2 * p \neq 0$!

If such properties were used in proving the correctness of a program, the programmer might be in for a rude surprise. Similar peculiarities arise with numbers near to one.

It should be noted that Pascal provides no double precision arithmetic.

	Pascal	Algol 3	Algol 4	Algol 68	Extended Fortran	MNF
q	K'	K'	K	K'	K"	
q = 0	F	F	F	F	F	
1 * q	K'	K'	K	K'	K"	
(1*q) = 0	F	F	F	F	F	
p	0'	π'	π	0'	0	0
p = 0	F	F	F	F	F	F
1 * p	0	0	0	0	0	0
(1*p) = 0	T	T	T	T	T	T
2 * p	0	0	0	0	0	K
(2*p) = 0	T	T	T	T	T	F
p + p	K'	K'	K	K'	K	K
(p+p) = 0	F	F	F	F	F	F
1 * (p+p)	K'	K'	K	K'	K	K
1 * (p+p) = 0	F	F	F	F	F	F
C	$1 - \epsilon''$	1	$1 - \epsilon$	1	$1 - \epsilon$	1
C = 1	F	T	T	F	T	T
C * C	$1 - \epsilon'$	1	$(1-\epsilon)^2$	$1 - \epsilon'$	$(1-\epsilon)^2$	1
C * C = 1	F	F	F	F	F	T

4. DOCUMENTATION

There must exist precise and readable documents describing the language and the implementation. There must be a rigorous definition of the language for reference, and there must be introductions for beginners. The implementation manuals must clearly describe the interface with the operating system, restrictions, extensions, and other deviations from the stan-

dard language, and implementation decisions relating language features to the machine. All information necessary for use must be in the manual, and the user must not have to experiment to determine facts about language features. The documentation must be readily available, whether this be through bookstores or manufacturers' representatives.

4.1. Pascal

There is a good user's manual and defining report [1]. It appears to correspond closely to the implementation on the Cyber, and clearly distinguishes between the Standard Pascal language and implementation quirks. It is on sale to the public through normal channels.

It is usually clear and explicit, except for a few guilty secrets. Several violations of run-time security are mentioned in this paper in the section on "Reliability", but the manual nowhere mentions that the language misuse that leads to such insecurity is indeed unlawful. Apparently it hopes that failing to mention an unchecked restriction will prevent users from running into it by accident.

The discussion of separate compilation in the Pascal user's manual can only be called inadequate. A few hints are given, and the bright thinker who is familiar with the CDC Cyber and the way things work there is then left to puzzle it out himself.

[13] is essential if one wishes to know the limitations of the arithmetic as implemented. It is unfortunate that these limitations are not clearly presented in chapter 13 of the Pascal user manual, which describes peculiarites of the Pascal 6000-3.4 implementation.

4.2. Algol 68

The documentation available at present is not extensive. The defining Report is an utterly precise definition of the language (except for its errors), but it is intended for language specialists, such as implementers, and it is not readily comprehensible to the ordinary user. A number of introductions to the language have sprung up, such as [4,12,20,21]; it is expected that more will follow.

The present CDC documentation describes the deviations from the stan-

dard language, and gives information about matters such as separate compilation, input/output, extra standard identifiers, and control cards. It is not always clear, and occasionally appears to suffer from excessive brevity.

4.3. Fortran

A ridiculously large number of textbooks on Fortran are available. Most of them describe dialects of Fortran without mentioning that they are in fact dialects. Few of them mention that there is a Fortran standard and fewer yet indicate which features are standard.

There exists a readable book describing the standard which contains a reprint of the standard [10].

4.4. Algol 60

The CDC documentation for Algol 3 and for Algol 4 consists of manuals containing

- a reprint of the Algol 60 Report [5], with inserts in a different type face describing changes made to the language, and giving further details on machine-dependent matters.
- a reprint of the ACM (Knuth) input/output proposal [14], again with inserts.
- a description of the various control cards involved, with explanation.
- an incomplete list of error messages.
- a description of the internal run-time organization, with bit maps for the various code words used.
- a description of the main processes involved in compilation.

In addition, many readable textbooks on Algol 60 exist, and some can usually be found in any technical bookstore.

5. EXPRESSIVE POWER

"Expressive power" is the most important (and most qualitative) aspect of programming language design. It refers to the interaction between

the language, good patterns of thought, and the domain of application. It is slowly becoming clear that one's programming languages determine one's patterns of thought, limit one's ability to see elegant methods of solving problems, and limit the useful generality and flexibility of one's programs. In general linguistics, the effect of language on thought is very difficult to distinguish from the effect of thought on language. This is different from computer linguistics for several reasons. First, a programming language is a relatively static entity, and does not change whenever a programmer discovers a new programming concept. A natural language usually responds instantly by acquiring a new word. Second, the class of programming language users is enormously larger than the class of language designers; a programmer has much less influence on his programming language than a speaker has on the natural language spoken in his circle of friends.

We shall examine expressive power from the viewpoint of structured programming and general purpose languages.

A programming language must be able to express the structure of programs written in it. The structure must be visible in the program, and not merely hidden in the mind of the programmer.

"General purpose" will be understood in the following sense. It must be possible to adapt the programming language to various purposes, perhaps by the definition of procedures and data types or by the choice of variable names. A large program usually contains collections of primitive routines that implement basic operations on those special kinds of objects that the program deals with. Such a collection of primitive routines in effect defines a specialized dialect of the programming language for the problem at hand. It is necessary to be able to build such specialized dialects onto a general-purpose language. There are many possible dialects for many different applications. Nonetheless, it is possible to distinguish some "general purpose" features. These are features which occur in many different dialects, or which are necessary tools for constructing dialects. A general purpose language must possess such features. The language designer should keep them down to a small, easily understood set. Because, ultimately, all operations are carried out on a computer, machine operations common to many computers are usually included in general-purpose languages.

It is not necessary, from the viewpoint of expressive power, that the features provided be easy to implement. It is important that they be easy to use and have simple properties. Many implementers unnecessarily complicate the properties of the primitive concepts of their language by propagating machine-dependent patterns of thought upwards. This can cause much agony to a programmer who finds himself required to think on two levels of abstraction at once - that of his dialect, and that of the machine hardware. These machine-dependent aspects often involve capacity constraints - limits on the size of a program, on the number of blocks or identifiers, and so forth. It is extremely important that there be no such hard limits. Such limits are usually imposed because an implementation has chosen fixed size tables or has chosen to place certain information in main storage, which is limited in capacity. It is important that all such limits be soft. Other implementation techniques should be invoked automatically when the limits are exceeded. Excess table information can be placed on disk or extended core storage, excess object code can be handled by overlay techniques, etc. This will probably influence efficiency, but not (directly) possibility. The price can very well be worth paying if it makes it unnecessary to confusingly and perhaps catastrophically maim a program in order to make it fit after a restriction has been encountered.

5.1. Pascal

At first sight, Pascal seems to be singularly free of the barnacles usually found encrusted on a programming language. Further inquiry, however, leads one to conclude that the ragged collections of extra features that other languages bear have been replaced by ragged and inconvenient restrictions.

The most important restriction in Pascal is that the sizes of all arrays are determined at compile time. It is therefore impossible to write many programs efficiently and clearly in such a way that they are independent of the amount of data to be processed. The only way to maintain a program library of, say, numerical routines is to keep it in source form. To use a routine in the library, the user must make a copy of the source

code, tinker with the array bounds, and include it in his own program. If he wishes to call the library procedure several times, giving it arrays of different sizes, he must include multiple copies of the procedure, each with a separate name and a separate array size. In this respect, Pascal is more restrictive than even Fortran, which at least permits a subroutine to be told by its caller what the size of an array argument is. Needless to say, algorithms which rely on a procedure that recursively calls itself for subarrays or smaller arrays than the original parameters can not be cleanly expressed in Pascal.

It is possible to parameterize array sizes at compile time, using a manifest constant. If this is declared once, its name can thereafter be used in array declarations, and the compiler will find the appropriate actual size at compile time. This makes it possible to localize the dependency of a program on array sizes. Unfortunately, expressions such as N + 1, where N is a manifest constant, or even 3 + 1 are not allowed as array bounds.

There are a host of restrictions on parameters and values yielded by procedures. One can divide values into two classes: "normal" values, and "second rate" values. Normal values are those which fit into one word on the CDC Cyber 73 (so much for machine independence), and second-rate values are those which do not. In the Pascal Report, when one reads through the various rules and restrictions, one finds that the second-rate values are records and arrays. There may be some sense in making such a distinction between elementary and compound values. On the other hand, Pascal presumably does not have double precision arithmetic because double precision values would have to be elementary but do not fit into a single machine word. By experimenting with the compiler one discovers that the type *alpha*, which is a packed array of characters, can often be used as if it were elementary after all! It does fit into a single word on the Cyber.

With the "normal" values, one can do anything one pleases. One can pass them to procedures as parameters, and one can return them as values. One cannot do this with the "second-rate" values. As an example of the elegance of Pascal's data structures, the Pascal manual shows how complex numbers can be represented as records containing two real numbers each. It is clear, since Pascal does not have complex numbers built in, that one

cannot use the usual operations +, -, *, and / on them, and procedures must be written. At this point the manual forgets about the example and goes on to other matters. One would expect to have to write functions *add*, *subtract*, *multiply*, and *divide* to perform arithmetic, so as to be able to write an expression

add (*multiply* (*a*, *b*), *multiply* (*c*, *d*))

instead of *a* * *b* + *c* * *d*. Unfortunately, (and here comes the catch) these procedures cannot be written either, since they would have to deliver second-rate values as function values.

It is not clear what the language or the programmer gains from such inconvenience. It cannot be efficiency, since the programmer who needs these facilities is now required to go to complex circumlocutions to express what might have been simple. Since the compiler can easily distinguish between single-word values and multiple-word values at compile time, the nonuser of multiple-word values should not need to suffer inefficiency for a feature he does not use.

"Power sets" are provided as one of the means of constructing new types from old. Given any scalar type (except *real*, which is a kludge), one can construct its power-set type, whose values are sets of values of the original type. This is a very clean concept of wide generality. Unfortunately, power sets are classed as normal values and must therefore fit within one CDC machine word. This, in turn, makes it quite clear that the purpose of introducing power sets was not to make available a clean and elegant concept for program construction, but to provide access to the underlying hardware bit manipulation. A power set of characters, for example, would seem to provide an elegant way of classifying characters. Instead, it is useless, because the character set contains 64 characters, and not 59. (Power sets are actually restricted to 59 bits instead of to 60 to avoid having to distinguish between positive and negative zero.)

In general, it does not pay in language design to place implementation restrictions to prevent certain "inefficient" features, if this forces the programmer who needs them to go to even more inefficient circumlocutions to compensate. The only time that such a restriction can be excused is if the unrestricted feature would cause significant costs to nonusers.

Pascal does provide something resembling Cobol and PL/I style record-directed input-output. A file consists of elements of some single data type; another file may have another data type. This type may be a record type, and it may be an array type. Each input or output operation transfers one value of the specified type, without formatting or conversion. Unlike Cobol, Pascal does not use the Record Manager.

Special kludges are provided to graft page and line structure onto character files, and to provide a small amount of formatting on output. Unlike Cobol and PL/I, Pascal does not provide any types for decimal arithmetic, and therefore the record structure cannot be used to achieve formatting.

Here are some stupid restrictions:

Power sets may have only 59 elements.

These 59 elements must each be such that *ord(element)* is between 0 and 58, inclusive.

Strings may be compared only if their length is less than 10 or a multiple of 10.

Only the first 10 characters of an identifier are significant.

5.2. Algol 68

The expressive power of Algol 68 is adequate for normal, and much abnormal, programming. It obtains this power from a reasonably well-chosen set of primitive concepts that can be combined in an extremely free manner. Restrictions have been placed on combinations that might be considered meaningless or dangerous, but no restrictions of concept have arisen from machine-dependent considerations.

Here are some of the primitive facilities it provides:

- basic data types - integer, real, character, boolean, bits, bytes. Integers, real numbers, bits, and bytes can each be of various "lengths", corresponding to various precisions that may be available on real machines.
- compound data types - structures (like Pascal records), arrays with bounds determined at run-time, pointers, procedures, and discriminated unions.

- a "heap" discipline of storage allocation, as well as a more conventional block-structured stack. The heap is garbage-collected.
- the ability to define new data types in terms of old ones.
- the ability to redefine most ordinary operators, and to define new ones (the example of complex arithmetic in Pascal can be done properly in Algol 68, even to the point of using the symbols "+", "-", "*", and "/" to denote the operations. On the other hand, Algol 68 already provides complex arithmetic, so the exercise, in this case, is academic).
- parallel processing.
- formatted, unformatted, and binary input/output.

The syntax and semantics for control structures are slightly better than in Algol 60 and Pascal. The most notable feature is the presence of "closing words", such as od at the end of a loop, and fi at the end of a conditional clause. These extras enable pairs of words such as do - od and if - fi to be used as brackets, and eliminate the vast majority of begins and ends present in Algol 60 and Pascal. The result is a more readable program. It is always clear in a syntactically correct program that a fi terminates a conditional clause; it is not always clear in Algol 60 which begin should be paired with any given end. The presence of different kinds of brackets makes visual matching easier.

The expressive power of Algol 68, taken as a whole, must be regarded as clearly superior to Fortran, Algol 60, and Pascal. However, there are deficiencies.

There is no true record input/ouput, as pioneered in Cobol and propagated in PL/I, and to some measure placed into Pascal. With true record input/output, the programmer specifies exactly how an input/output record should appear on the file, and a single input or output statement suffices to read the entire record into a group of variables. The layout of the record inside the machine is the same as that on the file - the formatting specification specifies how the variables are to be placed and accessed in main store, and not how information is transformed during input/output. Even with binary transput in Algol 68, it may be necessary to remap information, possibly rearranging it in a different order than that in which it resides in main store. Mixing character data and binary data on an Algol 68 file

is a dubitable matter; it is standard practice in Cobol and PL/I.

According to the Algol 68 Report, parallel processing is available. A programmer may divide his calculations among several independent parallel processes, which may even be run on separate CPU's. These processes may synchronize their occasional communications by means of special synchronization operators. Because of valid technical reasons connected with garbage collection and the Cyber operating systems, the CDC implementation does not provide true parallel processing. The facility therefore does not aid one in speeding up the work by harnessing more CPU's; it does aid in expressing algorithms clearly that require several independent parallel computations. The CDC implementation is nonetheless in compliance with the letter of the Report on this matter.

"Flexible" arrays constitute a seductive feature that is quite useless in practice. Algol 68 provides two kinds of arrays - "flexible" and "inflexible". The size of an inflexible array variable is fixed (at run time) when the variable is created. After an inflexible array variable is created, it always retains its own constant size. It is of course still possible to construct other array variables of different sizes if this is desired (perhaps when a block is re-entered). The size of a "flexible" array variable is not fixed when the variable is created, but can change anytime that a new array is assigned to the array variable. The array variable will change its size only when an entire array is assigned to it; however, any attempt to assign to an individual element or group of elements will be checked against the array bounds in the usual way. It is a common misconception that a new element can be inserted into a flexible array simply by assigning a value to a previously non-existent element. This is not so. In practice, flexible arrays are nearly useless, and they could probably be left out of the language with little loss.

Formatted input/output in Algol 68 is an incredibly complex subsystem which the casual user is advised to stay away from. It operates in the same style as the CDC Algol 60 formatting. In most cases, the desired results can be obtained much more easily, more efficiently, and more clearly by using unformatted input/output and the separate routines *whole*, *fixed*, and *float* provided by the language.

5.3. Fortran

In comparison with modern programming languages, Fortran must be considered woefully lacking in expressive power. Its essential form was fixed in the 1950's, and despite changes, it is still essentially the same language as then conceived.

Fortran is seriously lacking in convenient control and data structures, and it is difficult to break a program into parts, since the peculiarities of COMMON storage make the use of global variables difficult and hazardous. Subprograms may not call themselves or each other recursively. This makes many algorithms quite difficult to code.

Fortran was intended for programs involving simple repetitive numerical calculations, especially those involving matrices of fixed size. Relative to the state of programming language technology in the early and middle fifties, Fortran was a reasonably well-built product. Attempts to use Fortran for complicated problems or outside of its intended application area often lead to significant inefficiency and obscurity.

On the other hand, CDC has extended Fortran to make available many of the facilities of the Cyber hardware and of the Scope operating system. Fortran is therefore often used for very small problems primarily involving communication with the operating system.

The following is a list of various restrictions found in the Fortran Extended Reference Manual:

At most 10 characters may be stored in an integer.

At most 7 characters in an identifier (A.N.S. Fortran specifies 6).

DO loop indices must be less than 131072.

DO loops may be nested 50 deep.

At most 70 characters in a stop string.

At most 63 parameters to a subprogram or statement function.

At most 3 subscripts for an array.

At most 131071 words in a COMMON block.

Maximum field width is 131071.

At most 6 characters in a file name.

At most 50 files.

Record length at most 131071.

At most 125 labelled common blocks.

Unit numbers must be between 1 and 99 inclusive.

5.4. Algol 60

The expressive power of Algol 60 must be rated good, within limits. The control structure is adequate, and lends itself to comprehensible code. The data structures are adequate for numerical processes involving arrays (unlike Pascal, whose compile-time array bounds present severe difficulties). On the other hand, if the structure of the data does not fit well into arrays of numbers, the data structures of Algol 68 and Pascal must be judged superior.

Algol 60 has a "dangling else" difficulty. It is possible to leave out the else part of a conditional if it is empty. In certain nested conditional statements, such as if ... then if ... then ... else ... , this causes ambiguity, since it is not clear which if the else belongs with. Algol 60 makes an arbitrary choice here; it may not be that intended by the programmer.

We now present two short lists of stupid restrictions. They were obtained by paging through the Algol 3 and 4 reference manuals. The absence of a restriction from one of the two lists does not mean that the feature is not restricted, but simply that the restriction was not found in the manual. Restrictions which seem especially dangerous have been marked with asterisks.

The first list pertains to Algol 3:

Maximum length of identifiers: 256.

* At most 2383 different identifiers in a compilation (but the identifier table is usually full earlier).

Maximum depth of block nesting: 32.

At most 63 parameters for a procedure.

At most 20 subscripts for an array.

* At most 511 segments of object code of 512 words each.

Breakpoints cannot be used in segmented mode.

Nesting of blocks and compound statements at most 96.

* At most 50 separately compiled procedures in segmented mode.

At most 131072 words of object code per compilation.

Maximum replication (in a format): 262143.

At most 24 z's and d's before the exponent part in a number format.

At most 4 z's and d's in the exponent part.

At most 136 characters in a format item after replication.

At most 30 variables in a call to *FORMAT*.

R-L > 21 for the right and left margins of an output file.

It might be worth mentioning that the segmented mode is not available at SARA. Algol 3's segmented execution works only on certain obsolete versions of the Scope operating system. However, the current system has a loader with a different method of performing segmented loading of relocatable object code, which has nothing at all to do with the Algol 3 "segmented" option.

The following restrictions apply to Algol 4:

Maximum length of identifiers: 63.

* At most 4000 different identifiers, although the table is usually full earlier.

* At most 253 blocks.

Static block nesting at most 63.

At most 63 parameters per procedure.

* At most 131072 words of object code per compilation.

At most 63 subscripts for an array.

Maximum replicator: 262143.

At most 24 z's and d's before the exponent part in a number format.

At most 4 z's and d's in the exponent part.

At most 136 characters in a format item after replication.

6. LARGE PROGRAMS

When large programs are written, or when small programs become large (they inevitably do), serious logistic problems arise. The first difficulty is that the program itself becomes difficult to understand because of its complexity. At still greater size, it becomes difficult merely to deal with the amounts of text involved.

To deal with these problems, programming languages and their implementations have adopted various small-scale and large-scale structuring facilities and shoehorns. These may involve:

- pleasant control structures, such as the if-then-else of Algol 60 and loops.

- the ability to break a program into modules, such as procedures or groups of declarations.
- the ability to use textual layout to indicate program structure (e.g. indentation and pagination).
- the ability to restrict the scopes of names to those portions of the text where they are meaningful.
- the ability to code large modules containing smaller modules.
- the ability to compile modules separately, and to combine them subsequently.
- the ability to manage complicated file structures containing source and object code in an intelligent manner.
- shoehorns (such as overlay mechanisms) to handle object code or data which is too large for the address space of the machine.

Alas, any program management facilities that make vital use of the file system on direct-access secondary storage have to be considered useless at SARA, because of the policy of scratching files after four days of inactivity.

It is important to notice that it should be possible to fragment programs into separate compilations without having planned it beforehand, and without extensive rewriting. An unexpected split may become necessary through slow and gradual growth of an originally small program, or through importing programs from a larger installation.

The use of separate compilation and other shoehorn mechanisms should not exclude the use of other implementation facilities, such as run-time debugging tools. It is precisely when a program is large that one needs all the debugging aids one can get.

6.1. Pascal

Two methods are available for managing large programs.

First, procedures can be declared within one another, subject to the usual nested name scope rules. Unfortunately, this block structure does not permit declarations within begin-end blocks. The only "blocks" for determining the scopes of names are procedures. Declarations can be made within each procedure, and are then valid throughout the procedure. Such

nested procedure structure is adequate up to a fairly large program. It tends to break down only when the program itself becomes physically hard to manage.

Secondly, groups of procedures can be compiled separately. It is possible (but not easy) to surmise from the Pascal manual how this is to be done. The main program, which calls the procedures, is provided with dummy declarations of the separately compiled procedures. Such a dummy declaration is just like a normal declaration except that the body is replaced by the singly reserved word *extern* or *Fortran*. If *Fortran* is coded, the separately compiled program is called using Fortran linkage conventions (and it can thus be a Fortran routine; see the section on escape for complaints), and if *extern* is used Pascal linkage conventions are used.

To compile the external routines, a program is compiled with the "E+" option. The object code for each procedure will then have an entry point name consisting of the first seven letters of the procedure name. If the dummy procedure in the calling program has the same name, contact is achieved.

When one attempts to use this mechanism, however, one begins to feel like a sneak thief, relying on his wits and good luck to keep things from going wrong.

First, the E+ option should normally not be used if one is not interested in separate compilation. If two procedures happen to have the same name (which is legal if they are in different ranges) they will get the same entry point, and the system loader will refuse to load more than one of them. All calls will be routed to this single one, regardless of the program block structure.

Thus (as hinted in the manual) one must use the E- option, whereupon "a unique symbol is generated by the compiler" for each procedure. This would seem clear. However, if one is concerned with separate compilation, new phenomena occur. If one compiles a group of procedures separately, one might expect that one can use E+ for some of them to make them available publicly, and E- for others in the group that are to be available internally (perhaps one of them is a local procedure within a larger public one). However, as soon as one attempts this, the compiler begins

to generate nonunique names, contrary to promise. They are unique within the separate compilation, true, but they are the same "unique" external names generated by the compiler in the compilation of the main program and in every other separate compilation.

The only way to avoid this is, despite the apparent block structure of Pascal, to give each procedure in the separate compilation a unique name and specify E+ for all of them. No checks are provided by the compiler for duplicate external names. It is even possible to confuse procedures in disjoint blocks, with different static nesting depths. Furthermore, none of these names may be of the form "PRCdddd" (where each "d" stands for a digit) (these are the names Pascal generates).

There is no check on parameter compatibility between separately compiled procedures. The separate compilation method can, with care, be used in building large programs. However, it is virtually useless when building program libraries because of the array-bound restrictions. As lamented in the section "Expressive power", all Pascal array bounds are fixed at compile time. It is therefore impossible to precompile procedures for program libraries for array manipulation without knowing what the users' array sizes are going to be (before the users have even thought of the problem they are going to use the library for).

The Pascal compiler is capable of compiling itself. It itself is a Pascal program of some 5,000 lines.

6.2. Algol 68

The CDC Algol 68 compiler has already been used on a program of 2083 lines. To compile this program takes 83 seconds of CPU time on the CYBER 73 The program was not divided into separate compilations, although this would have been possible. No difficulties were encountered that would indicate any inherent capacity limits of the compiler. The compiler used approximately 77000 (octal) words of memory during compilation. Since the compiler is new, there is little experience with larger programs.

The separate compilation mechanism seems quite adequate for the development of some kinds of single large programs, but there are drawbacks and there is no adequate program library facility. In order to break a program

into parts to enable separate compilation, one must make a "user prelude". A user prelude much resembles a program, except that a special marker is placed in its outer range to indicate the placement of a not-yet-provided "main program". In addition, procedures in procedure declarations in the outer range may be replaced by place markers in another way. The compilation of this prelude produces two files, an object file and a symbol table.

The missing procedures and the missing main program may then be compiled separately as often as desired, if the symbol table is provided to the compiler as well. This symbol table provides information about global indicators. Full unrestricted use of global identifiers is possible, and type-checking occurs at compile time.

It is also possible to compile a further prelude in the hole provided for the main program.

This method is quite acceptable as a means for dividing a large program into pieces; it is not, however, adequate as a means for maintaining program libraries. Each such library will have to be compiled as a prelude. If they are compiled independently, they will cause the same storage to be allocated for their global variables; if they are compiled together they cannot be used separately.

It is to be hoped that CDC will eventually provide a proper library facility for their Algol 68 compiler.

6.3. Fortran

The only program structuring tools provided by Fortran are the DO-loop, the subprogram, the logical IF statement (for one-statement conditionals), the arithmetic statement functions, and the COMMON block. These have to be considered inadequate. Other small-scale tools are needed for conditional execution and for other kinds of loops. Algorithms involving these methods have to be encoded in a cumbersome way involving GO TO statements. If more than three or four such constructions are used in a single subprogram, it tends to become difficult to understand, and more subprograms must be used.

This in itself would not be so serious if adequate communication were possible between clearly identified groups of subprograms. Unfortunately, there is no means of grouping subprograms together into larger modules,

and the only means of sharing values between subprograms is to use parameter lists or COMMON blocks. Both mechanisms are highly error-prone. Since subprograms can be compiled completely independently of each other, no checks are performed to ensure type compatibility. If two subprograms do not agree as to the types or contents of parameters or COMMON blocks, the result is usually not an error message but complete chaos or wrong answers.

When a subprogram reaches about 200 lines, its internal structuring tends to break down; by the time a program reaches about 2000 lines, the hierarchy in its subprograms has usually become unclear.

Fortran Extended supports the same overlay mechanism as does Algol 4.

6.4. Algol 60

The conditional and loop control mechanisms and compound statements enable programs of 50 to 100 lines to be easily readable. Block structure and procedures extend this to 500-3000 line programs.

After this, no further syntactic aids are provided.

To enable truly gigantic programs to be compiled, separate compilation and libraries are available. To enable them to be executed, Algol 3 provides segmented execution, (but not with current versions of the Scope operating system, which can accomplish it independently of the Algol 3 "segmented" option), and Algol 4 provides overlays.

Unfortunately, separate compilation is more restrictive than in Fortran. In particular,

- only procedures may be compiled separately.
- there is no facility (such as Fortran's Common storage) to enable separately compiled procedures to share common global variables. Parameters can very quickly become unwieldy, and moreover, there may be only 63 of them.
- in Algol 3, numbers, not names, must be assigned to separately compiled procedures, and they must be referred to by number. This is unwieldy. Algol 4 permits names.

7. COMMUNICATION WITH THE OPERATING ENVIRONMENT

For a programming language to be useful for general use, it must have a decent operating-system interface. This means that those options in the operating system which might reasonably be expected to match with language features must indeed interact harmoniously. The most important areas involve fault detection, input/output, and interactiveness.

The implementation must cause errors detected by the operating system to be signalled to the programmer in a reasonable way. It must not give up on error recovery and readable post-mortem activity simply because the operating system has detected the error instead of the language implementation.

The implementation should be capable of accepting and producing the various kinds of files that the operating system supports. On the Cyber 73 under Scope 3.4, these supported file types are implemented by the "Record Manager".In addition, SARA provides an encoding for paper tapes as sequences of 12-bit characters. (Since line length is not always clearly defined on paper tape, it may be difficult to use the Record Manager for such files.) Furthermore, another form of 12-bit character files is recognized by some of the line printers as representing a character set of more than 64 characters. The Record Manager understands 6-bit characters only; it can therefore be difficult to recognize 12-bit characters. Most of the languages discussed here do not.

The implementation must be able to produce object code that is suitable for interactive use. Many implementations have a buffering problem: in a question-answer sequence between a user and the program, some systems require the user to provide the next line of input before giving him the response to his previous line. Furthermore, an interactive system often has difficulty with programs using large amounts of memory and exhibiting poor locality of reference. Since the CDC Cyber has no paging mechanism, poor locality of reference cannot be a problem; on the other hand, large memory consumption can significantly reduce system performance. At SARA, there is a limit of 60000 (octal) words of storage for an interactive program.

7.1. Pascal

All Pascal input/output files must reside on disk. Card reader and printer files are acceptable because the operating system, Scope, automatically spools them via disk. Magnetic tape files are not available to Pascal programs; they must first be copied to disk (and afterwards copied back) using a system utility. This causes the length of every tape record to be rounded upwards to a multiple of sixty bits, which may be tolerable on input, but probably not on output.

Pascal does not use the Record Manager. In one blow this eliminates most of the file types accepted by most other systems on the Cyber. Magnetic tapes would probably have been available if the Record Manager had been used: the problem is one of buffer length, and the Record Manager is willing to manage its own buffer length correctly.

The Pascal implementation recognizes two kinds of files: "character files" and "other files". Character files have the type "*file of character*"; the other files have the type "*file of othertype*", where "*othertype*" stands for some other type. Conceptually, Pascal sees these files as sequences of values of the given type, without arbitrary boundaries. On the other hand, character files are used for character input-output from the card reader and printer, and this imposes further structure on them. Now and then, between otherwise normal characters, an "end of line" or an "end of page" may occur. On output files for the line printer, the operating system requires the program to prefix a carriage-control character to each line to indicate whether it is to appear at the top of the page. This responsibility is faithfully handed over to the programmer by Pascal, and then forses him to treat print files differently from all other kinds of character files. (End of line already makes character files different from other files.) There is an obscure procedure called "newpage" mentioned in the Pascal manual, which is supposed to cause further output to begin on a new page; it is, however, not clear how it interacts with the programmer-supplied carriage-control character.

It is possible to get Pascal to read every bit of a disk file. To do this, one declares the file to be of a type which fills entire words evenly, for example:

file of packed array [0..59] *of bool*

file of packed array [0..4] *of* 0..4095

Other types are of course possible (subject to a number of peculiarities), and the choice of type can be used to provide some elementary structure for the file. The first type mentioned above will give easy access to individual bits; whereas the second provides twelve-bit bytes.

On the other hand, not everything reasonable will work. "*file of set of* 0..59", for example, is rejected by the compiler because it exceeds the maximum number of elements in a set (0..58 is allowed, but does not have the desired meaning). "*file of packed array* [1..15] *of* 0..255, which one might consider to indicate 15 bytes of eight bits packed into every two machine words (most 9-track tape files look like this) will not work, although it is proper Pascal. Pascal refuses to split an element of a packed array across a word boundary, and insists on leaving unused bits of padding in each word and using an extra word for the fifteenth byte. This completely defeats the purpose of the exercise. It may be said in defence of Pascal that its data structures were never *intended* to be used in this Cobol-like manner.

So-called "connected" files, which are "connected" to time-sharing terminals, suffer from a one-line lag. Pascal makes the end-of-file test available to the user before he reads the next (possible nonexistent) line. This is very reasonable. Unfortunately, Scope refuses to give end-of-file information until an attempt is made to read the possibly nonexistent line. Pascal therefore reads an extra line ahead internally. This is not objectionable in batch, but it is intolerable during time-sharing. It would have been better to wait for the programmer to issue the end-of-file test before reading ahead internally.

7.2. Algol 68

Algol 68 uses the Record Manager for its input and output, but it does not support all normal Record Manager file or record types. Only those kinds of files explicitly mentioned in the Algol 68 users' manual are supported. This appears to be because the Algol 68 implementation may move the input/output buffers during garbage collection, and then update the Record

Manager's tables accordingly. It appears, however, that the Record Manager occasionally maintains pointers other than those advertised in its documentation. The Algol 68 run-time system has not been debugged to handle input/output situations outside its specifications. Other record types may therefore work anyway if the Record Manager deals with them in a sufficiently uniform manner; this is not guaranteed.

The Algol 68 compiler and its object programs can both be run interactively under Intercom. The field length required fits into that tolerated at the SARA installations; however, since it requires more storage than the default field length, an EFL (Extend Field Length) must be given.

There is a peculiar incompatibility between the Algol 68 input/output system in the Report and that implemented. In most places where a programmer might expect to open a file by using *open* (according to the Report), the implementation requires that *establish* must be used instead. This is to inform the Algol 68 input/output of the maximum number of characters per line, the maximum number of lines per page, and the maximum number of pages. These must be provided by establish the first time that a file is accessed from a run of an Algol 68 program; thereafter the Algol 68 system remembers the data, and an *open* is required instead.

The Algol 68 system attempts to recover from operating-system-detected faults and to produce the normal diagnostic traceback anyway. Occasionally (such as when the escape mechanism is used to call a Fortran routine) it may fail to do this, presumably because of temporary non-adherence to its internal conventions.

7.3. Fortran

When working on the CDC Cyber, one rapidly gets the feeling that CDC in some way gives a preferred status to Fortran. New system features receive kludges for use with Fortran more often than with other languages. The other languages and compilers are then adapted to fit the Fortran conventions. Implementers of other languages therefore often provide a special interface for calling Fortran subroutines. This interface can fail if the Fortran subroutine attempts any input, since its input-output subsystem may not have been initialized.

Fortran, when calling subroutines, uses a Return Jump instruction, after placing the address of the parameter list in register A1 (and thus the address of the first parameter in register X1). The return jump instruction places the return address in the called program itself, and transfers control to the word after the word in which it places the return address. This calling sequence is the closest thing there is in Scope to a standard subroutine linkage convention. (It is, of course, useless for recursion; therefore, it is not used by languages which do permit recursion).

Fortran accepts the normal operating-system file structures, as defined by the Record Manager. It does not support the so-called 12-bit PE files for printing on an extended character-set printer. In fact, it uses the 6-bit 63-character display code that is normal in Scope.

CDC Fortran formatted input/output is as slow as it is in most Fortran systems. Extra nonstandard BUFFER IN and BUFFER OUT statements have been provided to enable programmers who are willing to do bit-fiddling to perform relatively fast and raw input and output.

The Record Manager has a special interface that enables its routines to be called directly using the Fortran linkage conventions. This is nonetheless somewhat awkward, since Fortran does not have convenient data-structuring facilities for describing the various system tables. Routines are therefore provided to fill in the various tables, given their addresses and strings describing the desired fields. Since storage allocation is left to the programmer, and is not checked by the system, the resulting communication can be tricky and insecure. Among the four languages we are comparing here, however, Fortran is the only one who provides complete access to the Record Manager, although it does it by completely bypassing the rest of its input/output system.

7.4. Algol 60

Algol 3 and Algol 4 have different interfaces with the operating system. They will therefore be discussed separately.

7.4.1. Algol 3

Algol 3 does not use the Record Manager. It processes Z-type files, with or without carriage control, which are the kinds of files that Scope uses for card reader, terminal, and printer input/output. Algol 3 provides a version of the so-called Knuth (or ACM) proposal, which handles formatting and provides procedures for input/output. In practice, The Algol 3 input/output system is abysmally slow, taking up unreasonable amounts of CPU time. It is not clear whether this is a fault of the implementation or of the specifications.

No random-access input/output is provided. The only thing remotely resembling it is a pair of procedures *WRITE ECS* and *READ ECS* for copying data between main store and ECS (extended core store).

When an Algol 3 program is used with a terminal, entire lines are transmitted as soon as they are complete. There is a one-line delay.

Algol 3 provides a segmentation mechanism for programs which would otherwise be too large for main memory. If this segmentation option is requested, object code is divided into segments, each of at most 512 words. There may be at most 511 segments. A segmented object program must be run under the control of the Algol 3 compiler, and then segments are loaded into memory as required, and removed again when the storage is required for other segments or for the run-time stack. This segmented mode is, however, not available under current versions of the Scope operating system. Instead, a segmenting option is provided by the normal relocating loader which has nothing to do with Algol 3's "segmented" option.

Since Algol 3 programs refer to files by channel numbers, some mechanism is needed to indicate the correspondence between channel numbers and files. This is the "channel card". An Algol 3 program requires one for each input/output file it uses (except for a standard input channel 60 for INPUT, and a standard output channel 61 for OUTPUT). The channel control cards do not form part of the Scope control card record; they are instead read from INPUT by the Algol 60 run-time system. For each channel used one must specify the channel number and its file name. One may also specify other parameters such as the maximum line width, the maximum page length, the number of spaces between numbers in standard format, the length

of physical records, buffering, whether the channel is to be used with the procedures *GET ARRAY* or *PUT ARRAY*, and the density and parity for magnetic tape.

Some of these parameters are parameters which can also be specified or respecified by the program itself.

7.4.2. Algol 4

Algol 4 uses the Record Manager for its input and output, and is therefore directly compatible with more of the recording modes available under Scope than Algol 3. Unfortunately, the channel card is still required to specify the correspondence between logical unit (channel) number and file names, but the other options are quite different. Most of the options on the channel card have been replaced by corresponding options on the Record Manager's file card, which does appear with the other control cards and not on the INPUT file. It is unfortunate that Algol 4 did not see fit to abolish the channel card entirely, although it does provide limited compatibity with Algol 3. The correspondence between channel number and file name could have been accomplished by the Record Manager LFN (logical file name) parameter. The other necessary leftovers are the length of a page (which the Algol 60 program can specify anew anyway), the presence or absence of carriage control characters, and the file type (word-addressible, indexed sequential, or sequential). It seems strange that a matter such as file type is not specified in the program but is left to a mandatory channel card at execution; if a user specifies it wrongly, the program would not be likely to function at all reasonably. It is not likely that the programmer would fail to know the file type when writing the program, because he must use different procedures for performing operations on files of different types.

The Algol 4 input/output system is also abysmally slow.

Algol 4 provides random access input/output in the form of word-addressible and indexed-sequential files. Unfortunately, the keys for the indexed-sequential file are restricted to integers.

Algol 4 provides an explicit planned overlay mechanism for segmenting large programs. Comments in various blocks identify the blocks as being overlays. Overlays form a tree structure with a nesting depth of 2.

8. ESCAPE

Sometimes it is necessary for a programmer to escape from the programming language in order to code a small part of the program in an other language. This is usually done

- in order to improve efficiency,
- in order to use subroutines already written in another language, or
- in order to gain access to system facilities not supported by the run-time system of the language.

It is clear that the practicality of a language implementation for large projects depends in part on the nature of the escape facilities, and on the frequency with which the escape facilities are necessary.

Such a method of escape is usually done by providing a mechanism for calling assembly language routines. Systems providing this usually advertise this as calling Fortran routines, in order to convince former Fortran users that they will not have to rewrite all their old subroutine libraries. An assembler language program then masquerades as a Fortran program by using the same linkage conventions.

Some compilers for high-level languages provide another scheme for escape. They permit the programmer to specify the machine instructions to be generated for each use of an operator or procedure when he declares the operator or procedure. This method is especially good for operations that can be implemented by a few machine instructions in line.

We also have the following questions:

- Does the implementation use normal operating-system interfaces for calling other subroutines?
- Does the operating system suggest any normal subroutine-linkage convention? If not, one can hardly blame the language.
- Can the programmer establish communication with the operating system concerning matters not or poorly built into the implementation?
- Does the language support the standard operating-system overlay mechanism?

8.1. Pascal

Pascal makes it possible to call subprograms written in Fortran. Such a Fortran subprogram must have a procedure declaration in the Pascal program, except that the procedure body is replaced by the reserved word "*Fortran*". It is of course possible to write the called subroutine in any language that supports Fortran linkage conventions appropriately, including Compass, the assembly language.

Unfortunately, there are some restrictions. If the Fortran subroutine expects a function or subroutine as parameter, then it must be provided a Fortran function or subroutine as parameter. It is not possible to provide a Pascal function; Pascal and Fortran have different linkage conventions, and neither is willing to provide the necessary interface for procedure parameters. One cannot even get around this by writing the alien procedure in Compass; the Pascal compiler perform a compile-time check to enforce the restriction.

If one has a Fortran routine which accepts an array of adjustable dimensions, as in

```
SUBROUTINE X(N, A(N))
DIMENSION A(N),
```

one must still declare it with fixed dimensions in the Pascal program. This is a direct consequence of the general Pascal restriction on array bounds, and it makes it impossible to call that one Fortran routine with Pascal arrays of different sizes from within one Pascal program. In this respect Pascal is actually less powerful than Fortran.

There is one other peculiarity with arrays as parameters. In Fortran, arrays are stored with the first subscript varying most rapidly; in Pascal, the last subscript varies most rapidly. Arrays are thus effectively transposed when Pascal hands them over to Fortran: A Pascal 6 by 4 array will be accepted in Fortran as a 4 by 6 array (transposed).

There are also peculiarities with complex numbers, since Pascal does not support them. In Pascal, a complex number must be represented as a record with two fields. Since it is a record, if it is passed as parameter it must be as a variable, and it cannot be yielded as a function value. This rather severely restricts the use of Fortran complex number library functions.

8.2. Algol 68

The Algol 68 compiler provides two means of escape.

The first is the declaration of a separately compiled routine which is written in another language. To call such a routine, it is necessary to provide the Algol 68 compiler with its entry-point name and the entry-point name of an interface module. An interface module is already provided for calling Fortran.

It is of course also possible to write Compass routines that directly conform to the Algol 68 linkage conventions.

The second means of escape is the so-called "ICF macros". The letters "ICF" stand for "Intermediate Code File". In the standard prelude, which defines all the basic Algol 68 operations and was itself compiled by the Algol 68 compiler, these ICF macros are used to provide the semantics for operations such as addition, and shift left. These ICF instructions resemble machine instructions, but do not specify registers or storage allocation. Each ICF instruction can be considered to produce a value when executed, and the programmer is provided with a means to specify which values are to be used in which later instructions. When a program contains a primitive operation which has been defined by an ICF macro, the ICF macro is expanded in-line into an intermediate code file. A later scan provides reasonably efficient storage and register assignment.

ICF macros can be used by the programmer instead of escape via separate compilation, although they appear to be intended as an internal compiler mechanism. If the programmer does this, he can in theory obtain code as efficient as that which the compiler itself produces, but there are drawbacks. Misuse of ICF code can impair reliability and error detection. The ICF macros are mentioned but not documented in the versions of the CDC Reference manual we have seen so far.

One serious problem with escaping to other programming languages is that Algol 68 takes over storage management within the entire user main storage area. It performs its own stack and heap administration within this area. Communicating with programs written in other languages (such as Pascal and Algol 60) which also manage storage within the user main storage area may therefore be practically impossible. The various run-time systems will engage in a storage-allocation war.

8.3. Fortran

Escape to machine and operating-system facilities other than those provided or consumed by Fortran is done by writing Compass subroutines with Fortran linkage conventions. Although no means are provided for calling Algol 60, Algol 68, or Pascal programs from Fortran, convenient means are nonetheless provided for calling Fortran programs from Algol 4, Algol 68, and Pascal programs. It is not possible to pass foreign-language procedures as parameters to Fortran subprograms and expect to be able to call them from the Fortran subprograms.

It is possible to bypass the Fortran input/output system and call the Record Manager directly from a Fortran program.

Fortran does support the operating-system standard overlay mechanism (as does Algol 4).

8.4. Algol 60

The Algol 3 separate-compilation facility can be used to escape from the Algol 3 system, by writing Compass programs that match the Algol 3 procedure calling conventions. These conventions are complicated. There exist Compass macros to reduce the effort, but these are also complicated. An entire chapter of the Algol 3 manual is dedicated to describing them. Since parameter passing and procedure calling are seriously expensive features in Algol 3, simply recoding a small routine in Compass is not likely to increase its efficiency significantly.

When the escape facilities are used with Algol 3, it is usually to perform input/output of an unusual nature or to gain efficiency. For example, NUMAL 3, a program library for numerical mathematics available from the Mathematical Centre, Amsterdam, uses Compass routines for elementary row operations on matrices. To use the Record Manager, it is necessary to use the escape mechanism. Algol 3 itself does not.

In Algol 4, the situation is different. Algol 4 provides a mechanism to specify that separately compiled routines use Fortran linkage conventions. Since the Record Manager and other system components are often written to match this linkage convention, escape to such other systems can be done more conveniently than from Algol 3. It is not necessary to write a

Compass conversion routine to translate Algol calls to Fortran calls; the Algol 4 compiler takes it on itself to use the proper linkage conventions.

The linkage conventions for Algol 3 and Algol 4 are incompatible.

9. COSTS

It is necessary that one be able to afford the language one uses. In some ways, the presence of a feature may make language use more expensive. since work may be necessary to implement it. Contrariwise, the implementer, with the hard machine at his disposal, may well do a better job than the language user who has only the programming language. Such a feature, if present and used, may reduce the costs from what they were without it. If one needs dynamic arrays, one needs dynamic arrays, no matter what kind of triangular hashed fixed-length tables the language may provide. If the language does not provide dynamic arrays, the programmer must construct his own for his own dialect, and he may do much worse than the implementer could have, because he does not have proper tools at his disposal.

It is therefore not possible to divide language features into "expensive" and "cheap" without considering the purposes for which they are used. Costs must be judged by the user, relative to his budget, his problem, and his experience.

It is of interest to indicate the cost imposed by various kinds of language use. These costs often arise from the overall quality of the implementation, instead of from any specific feature. It is also possible, however, that the demands imposed on the implementation by a specific feature increase the costs of other features, even when that specific feature is not used in a particular program. Garbage collection is a good example of this. The presence of a garbage collector for storage management in Algol 68 means that the rest of the Algol 68 system has to take appropriate precautions to ensure that the garbage collector can interpret storage layouts. The compiler cannot tell whether the garbage collector will actually be invoked by any particular program; therefore these precautions are always taken.

Four programs were written to provide information about costs:

Ackermann's function, to test recursion.

Cyclotomic polynomials, as a typical program.

Various input/output tests.

Feature timing, to give an approximate idea of the costs of various statements of the language.

Not all the timing measurements are equally reliable. A change of operating system occurred while the measurements were being made, and the clock on the new system (Scope 3.4.4) was much less accurate than the old one (Scope 3.4.1). The new clock has a tendency to remain stuck on a single value during many calls to the clock routine, and then to spontaneously jump to a new value by a step of between two to fifty-eight milliseconds (actual measurements!). There is a resulting uncertainty of this order of magnitude in every timing measurement performed under the new system. The only way around this is to increase the time the program takes to run, possibly by using a loop that executes it a hundred times, but this can get excessively expensive. (Since the above was written, SARA has replaced the 3.4.4 clock routine with the older, more reliable one. Clock jumps are now back down to one or two milliseconds).

All of the languages measured here use the same operating-system clock. They vary in their documentation, however, as to their claims to precision. Algol 68 claims to produce microsecond accuracy, Pascal and Algol 60 claim millisecond accuracy and Fortran claims centisecond accuracy.

Readers wishing to see additional statistics should consult [15] and [16].

The following abbreviations are used throughout this section:

ch	full subscript checking, or, in Pascal, range limit checking.
¬ch	No subscript checking.
pch	Partial checking, i.e., it is checked that final array-element addresses are within the array, but the individual subscripts are not checked against their individual bounds.

In addition, compiler options are occasionally mentioned in the form in which they must be specified to the compiler.

9.1 Ackermann's Function

Ackermann's function has been proposed by Wichmann [23,24] as a benchmark for systems programming languages. Its calculation by a straightforward algorithm involves much procedure calling and comparison, but little computa-

tion. The function is computed by a recursive procedure coded according to the following algorithm:

```
Ack (m,n) =
    if   m = 0 then n + 1
    elif n = 0 then Ack (m-1,1)
    else Ack (m-1, Ack(m,n-1))
    fi
```

Since Fortran does not provide recursion, it was left out of this test. The time per (recursive) call of the function and the number of words placed on the stack for each call were measured, the latter by examining core dumps.

system	options	average time per call (μs)	words per call
Pascal	ch	33	10
Algol 3	ch	478	21
Algol 4	ch	488	18
	¬ch,0=2,x=0	?	17
Algol 68	ch	63	7
	¬ch,z	62	7

These times were computed by measuring the time taken to compute Ack(3,i) for various values of i, and dividing by the (known) number of recursive calls this implies. The value of i was raised sufficiently high to force convergence of the average time; the above average times are precise to the microsecond.

9.2 A hundred and fifty cyclotomic polynomials

The various compilers were tested using a real program, at least one version of which had been originally written for a purpose other than that

of testing the compiler. The program choosen was one which symbolically computes and prints out the first one hundred and fifty cyclotomic polynomials. Various measurements were made on jobs compiling and running this program in various languages:

Job cost: the cost of the entire job, as judged by the SARA accounting routines.

Job CP: the amount of CP (central processor) time taken by the job.

compile CP: the CP time taken to compile the program, if reported by the compiler.

execution CP: the CP time taken to execute the program, as measured by the program itself.

calculation CP: the CP time taken to calculate the coefficients of the 150 cyclotomic polynomials.

IO CP: the CP time taken to format and print the answers.

compile FL: the amount of main storage required by the compiler to compile the program, if reported by the compiler.

The following abbreviations are used in the table:

ch full subscript checking.

pch partial subscript checking: the final array element must be within the array, but each subscript is not checked individually.

¬ch no subscript checking.

exec execution.

calc calculation.

The options actually given to the compiler to achieve various forms of checking are shown as well.

compiler	options	check?	job cost	job CP	compile CP	exec CP	calc CP	I/O CP	compile FL
Pascal	T+	ch	3.85	14.165	1.184	12.514	9.781	2.733	41412
	T-	¬ch	2.91	10.465	1.070	8.934	6.290	2.644	41412
A68		ch	18.18	61.259	6.125	51.813	31.870	19.943	53600
	Z	ch	17.97	60.566	7.096	50.237	30.807	19.425	53600
	A	¬ch	13.09	43,244	5.907	34.046	15.446	18.600	53600
	A,Z	¬ch	14.25	46.978	7.625	35.946	16.254	19.692	53600
Algol 3		pch	16.54	56.576	3.272	52.512	31.466	21.046	
	0	pch	17.03	58.032	4.708	52.516	30.900	21.616	
	N	¬ch	15.31	52.892	2.994	49.112	28.134	20.978	
	0,N	¬ch	15.25	52.426	4.094	47.544	26.527	21.017	
Algol 4	c=3[X=0]	ch	30.28	87.841	3.806	81.296	57.889	23.407	
	[X=0]	¬ch	16.20	46.165	3.837	39.487	17.174	22.313	
	0=1,X=0	¬ch	16.20	45.968	4.016	39.413	17.784	21.629	
	0=2,X=0	¬ch	15.35	43.340	4.069	36.435	15.288	21.207	
FTN	OPT=0,D	pch	17.94	45.583	1.910	40.888	36.470	4.418	
	OPT=0	¬ch	6.45	13.740	1.393	10.169	6.351	3.818	
	OPT=1	¬ch	5.30	11.036	1.564	7.333	3.785	3.548	
	OPT=2	¬ch	5.25	11.133	2.156	6.784	3.137	3.647	
MNF	D	ch	5.82	16.304	0.898	12.410	9.121	3.289	45200
		¬ch	4.48	12.306	0.596	8.414	5.444	2.970	45000
Algol 4	X=0,c=3,0=0	ch	30.38	88.223	3.872	81.721	57.808	23.913	
	X=1, 0=0	¬ch	18.31	52.462	3.990	45.979	23.593	22.386	
	X=0, 0=0	¬ch	16.19	46.153	4.062	39.447	17.302	22.145	
	X=1 0=2	¬ch	15.33	43.416	4.176	36.741	14.916	21.825	
	X=0,c=3,0=2	ch	(1.59)	(3.208)	compiler crash in pass 5				

The MNF compiler appears to perform subscript checking remarkably rapidly. This is because of a trick. If a subscript is a do-loop index, the subscript check may be performed on the limits of the do loop instead of on the subscript itself. The check is thus performed twice, once each on the initial and final limits in the do-loop, and it is not performed within the loop at all. This check will catch invalid programs as well as the more usual check on the subscript itself, and is much more efficient. Unfortunately, it will catch some valid programs as well, such as the following one:

```
      DIMENSION A(10)

      DO 3 I = 1, 50
      IF (I.GT.8) GO TO 6
      ..... A(I) .....
3     CONTINUE
      ...
6     ...
```

The versions of the program in the different languages are not identical, since the languages offer different possibilities. To make a fair judgement of a language, it is of course necessary to write the program in such a way that it fits the language. The language differences have effects on efficiency and on style; it is worthwhile studying the programs themselves as well as the execution times. The most notable differences are discussed below.

9.2.1. Algol 68

Polynomials are represented as arrays of coefficients by values of the mode pol, which is defined by

```
mode pol = flex [0:0] int.
```

This enables us to treat polynomials as single objects, and makes it possible to use only as much storage for a polynomial as is indicated by its degree. Individual coefficients can be altered in a polynomial without the cross-talk that might result if the mode ref [ ] int had been used instead. The operators "*" and "over" have been defined to operate on polynomials, so that the resulting expressions involving polynomial arithmetic will have the same form as normal arithmetic expressions.

In order to determine good points at which to break output lines, the standard procedure char number was used. This procedure reports the current position on an output line. If the program has come too near to the end of the line for another term of a polynomial to fit, it changes to a new line with pleasing indentation.

```
'BEGIN'     #CYCLOTOMISCHE POLYNOMEN#

      'REAL' P1:=CLOCK,P2,P3,P4;
      'MODE' 'POL' = 'FLEX' [0:0] 'INT';
      'INT' K=150;
      [1:K] 'POL' PHI;

      'PROC' F=('INT' N) 'REF' 'POL':                          # X**N - 1 #
            'BEGIN' 'HEAP' [0:N] 'INT' FX;
                    FX[N]:=1;FX[0]:=-1;
                    'FOR' I 'TO' N-1 'DO' FX[I]:=0 'OD';
                    FX
            'END' # OF F # ;

      'OP' *=('REF' 'POL' A,B) 'REF' 'POL':                    # A*B #
            'BEGIN' 'INT' N='UPB' A,M='UPB' B;
                    'HEAP' [0:M+N] 'INT' D;

                    'FOR' I 'FROM' 0 'TO' M+N 'DO' D[I]:=0 'OD';

                    'FOR' I 'FROM' N 'BY' -1 'TO' 'LWB' A
                    'DO' 'FOR' J 'FROM' M 'BY' -1 'TO' 'LWB' B
                         'DO' D[I+J]+:=A[I]*B[J] 'OD'
                    'OD';
                    D
            'END' # OF * # ;

      'OP' 'OVER'=('REF' 'POL' A,B) 'REF' 'POL':               # A/B #
                  'BEGIN' 'INT' M:='UPB' B;
                          'WHILE' B[M]=0 'DO' M-:=1 'OD';
                          'INT' J='UPB' A - M;
                          'HEAP' [0:J] 'INT' D;

                          'FOR' I 'FROM' 0 'TO' J 'DO' D[I]:=0 'OD';

                          'FOR' N 'FROM' 'UPB' A 'BY' -1 'TO' M
                          'DO' 'IF' A[N]/=0
                               'THEN' 'IF' A[N] 'MOD' B[M]/=0
                                      'THEN' PRINT("DELING GAAT NIET OP")
                                      'FI';
                                      'INT' K=A[N] 'OVER' B[M] , J=N-M;
                                      'FOR' I 'FROM' J 'TO' N
                                      'DO' A[I]-:=K*B[I-J] 'OD';
                                     D[J]:=K
                               'FI'
                          'OD';
                          D
                  'END' # OF 'OVER' # ;
```

```
        'FOR' I 'TO' K
        'DO' 'POL' G;G[Ø]:=1;
             'FOR' J 'TO' I 'OVER' 2
             'DO' 'IF' I 'MOD' J=Ø
                  'THEN' G:=G*PHI[J]
                  'FI'
             'OD';
             PHI[I]:=F(I) 'OVER' G
        'OD';

        P2:=CLOCK;

        P3:=CLOCK;

        PRINT((NEWLINE,"DE EERSTE ",WHOLE(K,Ø)," CYCLOTOMISCHE ",
                       "POLYNOMEN",NEWLINE));
        'FOR' I 'TO' K
        'DO'
             PRINT((NEWLINE,NEWLINE,"PHI",WHOLE(I,Ø),"    "));
             'FOR' J 'FROM' Ø 'TO' 'UPB' PHI[I]
             'DO' 'IF' PHI[I][J]/=Ø
                  'THEN' 'IF' CHAR NUMBER(STAND OUT)>128
                         'THEN' PRINT((NEWLINE,"          "))
                         'FI';
                         PRINT((WHOLE(PHI[I][J],Ø),"X",WHOLE(J,Ø),"  "))
                  'FI'
             'OD'
        'OD';

        P4:=CLOCK;

        'TO' 5 'DO' PRINT(NEWLINE) 'OD';
        PRINT(("REKENTIJD: ",FIXED(P2-P1,Ø,6)," SEC.",NEWLINE));
        PRINT(("UITVOERTIJD: ",FIXED(P4-P3,Ø,6)," SEC.",NEWLINE));
        PRINT(("TOTALE TIJD: ",FIXED(P4-P1,Ø,6)," SEC.",NEWLINE))

'END'
```

```
PHI103  1X0  1X1  1X2  1X3  1X4  1X5  1X6  1X7  1X8  1X9  1X10  1X11  1X12  1X13  1X14  1X15  1X16  1X17  1X18  1X19  1X20  1X21
        1X22  1X23  1X24  1X25  1X26  1X27  1X28  1X29  1X30  1X31  1X32  1X33  1X34  1X35  1X36  1X37  1X38  1X39  1X40  1X41
        1X42  1X43  1X44  1X45  1X46  1X47  1X48  1X49  1X50  1X51  1X52  1X53  1X54  1X55  1X56  1X57  1X58  1X59  1X60  1X61
        1X62  1X63  1X64  1X65  1X66  1X67  1X68  1X69  1X70  1X71  1X72  1X73  1X74  1X75  1X76  1X77  1X78  1X79  1X80  1X81
        1X82  1X83  1X84  1X85  1X86  1X87  1X88  1X89  1X90  1X91  1X92  1X93  1X94  1X95  1X96  1X97  1X98  1X99  1X100  1X101
        1X102

PHI104  1X0  -1X4  1X8  -1X12  1X16  -1X20  1X24  -1X28  1X32  -1X36  1X40  -1X44  1X48

PHI105  1X0  1X1  1X2  -1X5  -1X6  -2X7  -1X8  -1X9  1X12  1X13  1X14  1X15  1X16  1X17  -1X20  -1X22  -1X24  -1X26  -1X28  1X31
        1X32  1X33  1X34  1X35  1X36  -1X39  -1X40  -2X41  -1X42  -1X43  1X46  1X47  1X48

PHI106  1X0  -1X1  1X2  -1X3  1X4  -1X5  1X6  -1X7  1X8  -1X9  1X10  -1X11  1X12  -1X13  1X14  -1X15  1X16  -1X17  1X18  -1X19
        1X20  -1X21  1X22  -1X23  1X24  -1X25  1X26  -1X27  1X28  -1X29  1X30  -1X31  1X32  -1X33  1X34  -1X35  1X36  -1X37  1X38
        -1X39  1X40  -1X41  1X42  -1X43  1X44  -1X45  1X46  -1X47  1X48  -1X49  1X50  -1X51  1X52

PHI107  1X0  1X1  1X2  1X3  1X4  1X5  1X6  1X7  1X8  1X9  1X10  1X11  1X12  1X13  1X14  1X15  1X16  1X17  1X18  1X19  1X20  1X21
        1X22  1X23  1X24  1X25  1X26  1X27  1X28  1X29  1X30  1X31  1X32  1X33  1X34  1X35  1X36  1X37  1X38  1X39  1X40  1X41
        1X42  1X43  1X44  1X45  1X46  1X47  1X48  1X49  1X50  1X51  1X52  1X53  1X54  1X55  1X56  1X57  1X58  1X59  1X60  1X61
        1X62  1X63  1X64  1X65  1X66  1X67  1X68  1X69  1X70  1X71  1X72  1X73  1X74  1X75  1X76  1X77  1X78  1X79  1X80  1X81
        1X82  1X83  1X84  1X85  1X86  1X87  1X88  1X89  1X90  1X91  1X92  1X93  1X94  1X95  1X96  1X97  1X98  1X99  1X100  1X101
        1X102  1X103  1X104  1X105  1X106

PHI108  1X0  -1X18  1X36
```

9.2.2. Pascal

Pascal does not permit variable array bounds, and it was therefore necessary to use the maximum size of polynomials, 150, for all arrays which might contain polynomials. This, of course, causes a significant waste of storage. The rather large array of results is thereby doubled in size. The number of array elements actually used by a polynomial has been placed in element -1 of each array. This limit is used during multiplication and division to save execution time.

It is not possible to redefine operators in Pascal, and therefore procedures were written instead to perform mutiplication and division. These procedures produce their computed results in parameters, because Pascal prohibits yielding an array as function value.

A number of global variables have been declared. We would have preferred to declare these variables locally in the blocks where they are used, but Pascal does not permit this. The variable declarations have therefore been moved to the procedure headings.

Pascal does not provide a procedure to determine the current position in an output line. It was therefore necessary to maintain an independent counter to determine this. Producing output with reasonable layout was therefore marginally more difficult than in Algol 68.

```
PROGRAM CYCLPOL(OUTPUT);
(*$T-*)

(*CYCLOTOMISCHE POLYNOMEN*)

VAR I,P1,P2,P3,P4:INTEGER;

PROCEDURE HOOFDPROGRAMMA;

CONST K=150;
TYPE POL=ARRAY[-1..K] OF INTEGER;
VAR I,J,C,H:INTEGER; G,FX:POL; PHI:ARRAY[1..K] OF POL ;

PROCEDURE F(N:INTEGER;VAR P:POL);                                  (* X**N - 1 *)
          VAR I:INTEGER;
          BEGIN P[-1]:=N;P[0]:=-1;P[N]:=1;
                FOR I:=1 TO N-1 DO P[I]:=0
          END (* OF F *) ;

PROCEDURE MULPOL(VAR A,B:POL);                                     (* A:=A*B *)
          VAR I,J,M,N:INTEGER;
                C:POL;
          BEGIN M:=A[-1];N:=B[-1];
                A[-1]:=M+N;
                FOR I:=0 TO M DO C[I]:=A[I];


                FOR I:=0 TO M+N DO A[I]:=0;

                FOR I:=0 TO M
                DO FOR J:=0 TO N
                   DO A[I+J]:=A[I+J]+C[I]*B[J]

          END (* OF MULPOL *) ;

PROCEDURE DIVPOL(VAR A,B,C:POL);                                   (* C:=A/B *)
          VAR M,I,J,KK,N:INTEGER;
          BEGIN M:=B[-1];
                WHILE B[M]=0 DO M:=M-1;

                J:=A[-1]-M;
                FOR I:=0 TO J DO C[I]:=0;
                C[-1]:=J;
                FOR N:=A[-1] DOWNTO M
                DO IF A[N]<>0
                   THEN BEGIN J:=N-M;
                              IF A[N] MOD B[M]<>0
                              THEN WRITELN(' DELING GAAT NIET OP');
                              KK:=A[N] DIV B[M];
                              FOR I:=N DOWNTO J
                              DO A[I]:=A[I]-KK*B[I-J];
                              C[J]:=KK
                        END
          END (* OF DIVPOL *) ;
```

```
BEGIN FOR I:=1 TO K
      DO BEGIN G[-1]:=0;G[0]:=1;
               FOR J:=1 TO I DIV 2
               DO IF I MOD J=0
                  THEN MULPOL(G,PHI[J]);
               F(I,FX);
               DIVPOL(FX,G,PHI[I]);
         END;

         P2:=CLOCK;

         P3:=CLOCK;

      WRITELN('1DE EERSTE ',K:1,' CYCLOTOMISCHE POLYNOMEN');WRITELN;
      FOR I:=1 TO K
      DO BEGIN WRITELN;WRITELN;WRITE(' PHI',I:1,'    ');
               C:=10;
               FOR J:= 0 TO PHI[I][-1]
               DO IF PHI[I][J]<>0
                  THEN BEGIN IF C>128 THEN BEGIN WRITELN;C:=10;
                                             WRITE('           ')
                                        END;
                             WRITE(PHI[I][J]:1,'X',J:1,'  ');
                             H:=10;
                             WHILE J>=H DO BEGIN C:=C+1 ;H:=H*10 END;
                             IF PHI[I][J]<0 THEN C:=C+6
                                            ELSE C:=C+5
                       END
         END
END;

BEGIN P1:=CLOCK;
HOOFDPROGRAMMA;
P4:=CLOCK;
FOR I:=1 TO 5 DO WRITELN;
WRITELN(' REKENTIJD:  ',P2-P1:6,' MSEC.');
WRITELN(' UITVOERTIJD:  ',P4-P3:6,' MSEC.');
WRITELN(' TOTALE TIJD:  ',P4-P1:6,' MSEC.')
END.
```

```
PHI103   1X0  1X1  1X2  1X3  1X4  1X5  1X6  1X7  1X8  1X9  1X10  1X11  1X12  1X13  1X14  1X15  1X16  1X17  1X18  1X19  1X20  1X21
         1X22  1X23  1X24  1X25  1X26  1X27  1X28  1X29  1X30  1X31  1X32  1X33  1X34  1X35  1X36  1X37  1X38  1X39  1X40  1X41
         1X42  1X43  1X44  1X45  1X46  1X47  1X48  1X49  1X50  1X51  1X52  1X53  1X54  1X55  1X56  1X57  1X58  1X59  1X60  1X61
         1X62  1X63  1X64  1X65  1X66  1X67  1X68  1X69  1X70  1X71  1X72  1X73  1X74  1X75  1X76  1X77  1X78  1X79  1X80  1X81
         1X82  1X83  1X84  1X85  1X86  1X87  1X88  1X89  1X90  1X91  1X92  1X93  1X94  1X95  1X96  1X97  1X98  1X99  1X100  1X101
         1X102

PHI104   1X0  -1X4  1X8  -1X12  1X16  -1X20  1X24  -1X28  1X32  -1X36  1X40  -1X44  1X48

PHI105   1X0  1X1  1X2  -1X5  -1X6  -2X7  -1X8  -1X9  1X12  1X13  1X14  1X15  1X16  1X17  -1X20  -1X22  -1X24  -1X26  -1X28  1X31
         1X32  1X33  1X34  1X35  1X36  -1X39  -1X40  -2X41  -1X42  -1X43  1X46  1X47  1X48

PHI106   1X0  -1X1  1X2  -1X3  1X4  -1X5  1X6  -1X7  1X8  -1X9  1X10  -1X11  1X12  -1X13  1X14  -1X15  1X16  -1X17  1X18  -1X19
         1X20  -1X21  1X22  -1X23  1X24  -1X25  1X26  -1X27  1X28  -1X29  1X30  -1X31  1X32  -1X33  1X34  -1X35  1X36  -1X37  1X38
         -1X39  1X40  -1X41  1X42  -1X43  1X44  -1X45  1X46  -1X47  1X48  -1X49  1X50  -1X51  1X52

PHI107   1X0  1X1  1X2  1X3  1X4  1X5  1X6  1X7  1X8  1X9  1X10  1X11  1X12  1X13  1X14  1X15  1X16  1X17  1X18  1X19  1X20  1X21
         1X22  1X23  1X24  1X25  1X26  1X27  1X28  1X29  1X30  1X31  1X32  1X33  1X34  1X35  1X36  1X37  1X38  1X39  1X40  1X41
         1X42  1X43  1X44  1X45  1X46  1X47  1X48  1X49  1X50  1X51  1X52  1X53  1X54  1X55  1X56  1X57  1X58  1X59  1X60  1X61
         1X62  1X63  1X64  1X65  1X66  1X67  1X68  1X69  1X70  1X71  1X72  1X73  1X74  1X75  1X76  1X77  1X78  1X79  1X80  1X81
         1X82  1X83  1X84  1X85  1X86  1X87  1X88  1X89  1X90  1X91  1X92  1X93  1X94  1X95  1X96  1X97  1X98  1X99  1X100  1X101
         1X102  1X103  1X104  1X105  1X106

PHI108   1X0  -1X18  1X36
```

9.2.3. Algol 60

It is impossible to declare new data types or for arrays to have arrays as elements in Algol 60. The result is that polynomials have been placed as rows in a two-dimensional array. The -1st element of each row once again indicates the number of rows in actual use. To give a procedure a specific row of this array as parameter, the entire array is handed over, together with an integer specifying which row is meant.

The procedures for multiplication and division must once again yield their results in output parameters.

Unlike in Algol 68 and Pascal, in Algol 60 it is difficult to let the field width for printing a number depend on the number, even with "expensive" Knuthput. We therefore chose a fixed field width and used it for all exponents. The result is that the output is slightly less readable, since the "x" is separated from the exponent by spaces. Once more, we have used an independent counter to determine the position on the output line.

```
"BEGIN"

    "COMMENT" CYCLOTOMISCHE POLYNOMEN ;

    "INTEGER" K ;
    "REAL" P1,P2,P3,P4 ;

    P1 := CLOCK ;

    K := 150 ;

    "BEGIN"
        "INTEGER" I,J,C ;
        "INTEGER" "ARRAY" PHI[1:K,-1:K] ;
        "COMMENT" "CHECKON" PHI ;

        "PROCEDURE" F(N,P) ; "VALUE" N ; "INTEGER" N ;
        "COMMENT" X**N - 1 ;
                                "INTEGER" "ARRAY" P ;
                                "COMMENT" "CHECKON" P ;
            "BEGIN"
                "INTEGER" I ;
                P[-1] := N ; P[0] := -1 ; P[N] := 1 ;
                "FOR" I := 1 "STEP" 1 "UNTIL" N - 1 "DO" P[I] := 0

            "END" OF F ;

        "PROCEDURE" MUL(A,B,T) ; "VALUE" T ; "INTEGER" T ;
        "COMMENT" A:=A*B ;
                                "INTEGER" "ARRAY" A,B ;
                                "COMMENT" "CHECKON" A,B ;
            "BEGIN"
                "INTEGER" I,J,M,N ;
                "INTEGER" "ARRAY" C[0:A[-1]] ;
                "COMMENT" "CHECKON" C ;
                M :=A[-1] ; N := B[T,-1] ;
                A[-1] := M + N ;
                "FOR" I := 0 "STEP" 1 "UNTIL" M "DO" C[I] := A[I] ;

                "FOR" I := 0 "STEP" 1 "UNTIL" M + N "DO" A[I] := 0 ;

                "FOR" I := 0 "STEP" 1 "UNTIL" M
                "DO" "FOR" J := 0 "STEP" 1 "UNTIL" N
                     "DO" A[I + J] := A[I + J] + C[I] * B[T,J]

            "END" OF MUL ;

        "PROCEDURE" DIV(A,B,C,T) ; "VALUE" T ; "INTEGER" T ;
        "COMMENT" C[T, ]:=A/B ;
                                "INTEGER" "ARRAY" A,B,C ;
                                "COMMENT" "CHECKON" A,B,C ;
```

```
    "BEGIN"
        "INTEGER" M,I,J,KK,N ;
        M := B[-1] ; I := 1 ;
        "FOR" I := I + 1 "WHILE" B[M] = 0 "DO" M := M - 1 ;

        J := A[-1] - M ;

        "FOR" I := 0 "STEP" 1 "UNTIL" J "DO" C[T,I] := 0 ;
        C[T,-1] := J ;

        "FOR" N :=A[-1] "STEP" -1 "UNTIL" M
        "DO" "IF" A[N] ^= 0
             "THEN" "BEGIN"
                        J := N - M ;
                        "IF" A[N] // B[M] * B[M] ^= A[N]
                        "THEN" OUTPUT(61,"(""("DELING GAAT ")",
                                          "("NIET OP")""")");
                        KK := A[N] // B[M] ;
                        "FOR" I := N "STEP" -1 "UNTIL" J
                        "DO" A[I] := A[I] - KK * B[I - J];

                        C[T,J] := KK
                    "END"

    "END" OF DIV ;

"FOR" I := 1 "STEP" 1 "UNTIL" K
"DO" "BEGIN"
         "INTEGER" "ARRAY" G,FX[-1:I] ;
         "COMMENT" "CHECKON" G,FX ;
         G[-1] := 0 ; G[0] := 1 ;
         "FOR" J := 1 "STEP" 1 "UNTIL" I // 2
         "DO" "IF" I // J * J = I
              "THEN" MUL(G,PHI,J);

         F(I,FX) ;
         DIV(FX,G,PHI,I)
     "END" ;
```

```
            P2 := CLOCK ;

            P3 := CLOCK ;

            OUTPUT(61,"(""("DE EERSTE ")",2ZD,"(" CYCLOTOMISCHE ")",
                        "("POLYNOMEN")",//")",K);
            "FOR" I := 1 "STEP" 1 "UNTIL" K
            "DO" "BEGIN"
                     OUTPUT(61,"("//,"("PHI")",2ZD3B")",I);
                     C := 10 ;
                     "FOR" J := 0 "STEP" 1 "UNTIL" PHI[I,-1]
                     "DO" "IF" PHI[I,J] ^= 0
                          "THEN" "BEGIN"
                                    "IF" C > 129
                                    "THEN" "BEGIN"
                                              OUTPUT(61,"("/,9B")");
                                              C := 10
                                           "END" ;
                                    OUTPUT(61,"("+D,"("X")",2ZDB")",
                                                 PHI[I,J],J);
                                    C := C + 7
                                 "END"
                "END"
    "END" ;

    P4 := CLOCK ;

    OUTPUT(61,"("7/,"("REKENTIJD: ")",ZD.6D,"(" SEC.")"")",P2-P1);
    OUTPUT(61,"("/,"("UITVOERTIJD: ")",ZD.6D,"(" SEC.")"")",P4-P3);
    OUTPUT(61,"("/,"("TOTALE TIJD: ")",ZD.6D,"(" SEC.")"")",P4-P1)
"END"
```

```
PHI103  +1X  0 +1X  1 +1X  2 +1X  3 +1X  4 +1X  5 +1X  6 +1X  7 +1X  8 +1X  9 +1X 10 +1X 11 +1X 12 +1X 13 +1X 14 +1X 15 +1X 16 +1X 17
        +1X 18 +1X 19 +1X 20 +1X 21 +1X 22 +1X 23 +1X 24 +1X 25 +1X 26 +1X 27 +1X 28 +1X 29 +1X 30 +1X 31 +1X 32 +1X 33 +1X 34 +1X 35
        +1X 36 +1X 37 +1X 38 +1X 39 +1X 40 +1X 41 +1X 42 +1X 43 +1X 44 +1X 45 +1X 46 +1X 47 +1X 48 +1X 49 +1X 50 +1X 51 +1X 52 +1X 53
        +1X 54 +1X 55 +1X 56 +1X 57 +1X 58 +1X 59 +1X 60 +1X 61 +1X 62 +1X 63 +1X 64 +1X 65 +1X 66 +1X 67 +1X 68 +1X 69 +1X 70 +1X 71
        +1X 72 +1X 73 +1X 74 +1X 75 +1X 76 +1X 77 +1X 78 +1X 79 +1X 80 +1X 81 +1X 82 +1X 83 +1X 84 +1X 85 +1X 86 +1X 87 +1X 88 +1X 89
        +1X 90 +1X 91 +1X 92 +1X 93 +1X 94 +1X 95 +1X 96 +1X 97 +1X 98 +1X 99 +1X100 +1X101 +1X102

PHI104  +1X  0 -1X  4 +1X  8 -1X 12 +1X 16 -1X 20 +1X 24 -1X 28 +1X 32 -1X 36 +1X 40 -1X 44 +1X 48

PHI105  +1X  0 +1X  1 +1X  2 -1X  5 -1X  6 -2X  7 -1X  8 -1X  9 +1X 12 +1X 13 +1X 14 +1X 15 +1X 16 +1X 17 -1X 20 -1X 22 -1X 24 -1X 26
        -1X 28 +1X 31 +1X 32 +1X 33 +1X 34 +1X 35 +1X 36 -1X 39 -1X 40 -2X 41 -1X 42 -1X 43 +1X 46 +1X 47 +1X 48


PHI106  +1X  0 -1X  1 +1X  2 -1X  3 +1X  4 -1X  5 +1X  6 -1X  7 +1X  8 -1X  9 +1X 10 -1X 11 +1X 12 -1X 13 +1X 14 -1X 15 +1X 16 -1X 17
        +1X 18 -1X 19 +1X 20 -1X 21 +1X 22 -1X 23 +1X 24 -1X 25 +1X 26 -1X 27 +1X 28 -1X 29 +1X 30 -1X 31 +1X 32 -1X 33 +1X 34 -1X 35
        +1X 36 -1X 37 +1X 38 -1X 39 +1X 40 -1X 41 +1X 42 -1X 43 +1X 44 -1X 45 +1X 46 -1X 47 +1X 48 -1X 49 +1X 50 -1X 51 +1X 52

PHI107  +1X  0 +1X  1 +1X  2 +1X  3 +1X  4 +1X  5 +1X  6 +1X  7 +1X  8 +1X  9 +1X 10 +1X 11 +1X 12 +1X 13 +1X 14 +1X 15 +1X 16 +1X 17
        +1X 18 +1X 19 +1X 20 +1X 21 +1X 22 +1X 23 +1X 24 +1X 25 +1X 26 +1X 27 +1X 28 +1X 29 +1X 30 +1X 31 +1X 32 +1X 33 +1X 34 +1X 35
        +1X 36 +1X 37 +1X 38 +1X 39 +1X 40 +1X 41 +1X 42 +1X 43 +1X 44 +1X 45 +1X 46 +1X 47 +1X 48 +1X 49 +1X 50 +1X 51 +1X 52 +1X 53
        +1X 54 +1X 55 +1X 56 +1X 57 +1X 58 +1X 59 +1X 60 +1X 61 +1X 62 +1X 63 +1X 64 +1X 65 +1X 66 +1X 67 +1X 68 +1X 69 +1X 70 +1X 71
        +1X 72 +1X 73 +1X 74 +1X 75 +1X 76 +1X 77 +1X 78 +1X 79 +1X 80 +1X 81 +1X 82 +1X 83 +1X 84 +1X 85 +1X 86 +1X 87 +1X 88 +1X 89
        +1X 90 +1X 91 +1X 92 +1X 93 +1X 94 +1X 95 +1X 96 +1X 97 +1X 98 +1X 99 +1X100 +1X101 +1X102 +1X103 +1X104 +1X105 +1X106

PHI108  +1X  0 -1X 18 +1X 36
```

9.2.4. Fortran

As in Algol 60, the polynomials have once more been placed in an array. Because all Fortran arrays have a lower bound of 1 instead of -1, array elements have all been shifted over by two places. The first element of each row gives the upper bound of the array elements used.

Polynomial multiplication and division is once more done in subroutines, but this time the language requires array bounds to be passed as parameters, in order to use them in DIMENSION statements. In Fortran, it is less easy to localize dependency on the number of polynomials computed to one place in the program, since each array dimension must be specified as an integral constant, and a manifest constant such as Pascal uses is not permitted.

The routine for producing output is considerably more complicated than in the other three languages, since Fortran starts a new line for each output statement. The values to be printed must therefore be selected beforehand and placed in a buffer, so that an implied DO loop in an output statement can write them all neatly. The repetition mechanisms of the FORMAT statement can be used to determine proper line breakage. Field widths are constant again; it is as difficult to let the width of a number depend on its value as in Algol 60.

```
C$     DEBUG
C$     ARRAYS
       PROGRAM CYPO (OUTPUT,TAPE 6=OUTPUT)
C CYCLOTOMISCHE POLYNOMEN
       IMPLICIT INTEGER (A-O)
       COMMON/L1/P2,P3
       K=150
       P1=SECOND(T)
       CALL PROC(K)
       P4=SECOND(T)
       WRITE(6,3R P2-P1,P4-P3,P4-P1
     3 FORMAT(5(/),* REKENTIJD: *,F9.6,* SEC.*,/,* UITVOERTIJD: *,F9.6,*
      CSEC.*,/,* TOTALE TIJD: *,F9.6,* SEC.*)
        STOP
       END

       SUBROUTINE PROC(K)
       INTEGER G(152),FX(152),PHI(150,152),COEF(151),POW(151)
       COMMON/L1/P2,P3
       K1=K+2
       DO 50 I=1,K
          G(1)=2
          G(2)=1
          IF (I.EQ.1) GO TO 45
             I2=I/2
             DO 40 J=1,I2
                IF (I/J*J.NE.I) GO TO 40
                   CALL MUL(G,PHI,J,K,K1)
    40         CONTINUE
    45    CALL F(I,FX,K1)
    50    CALL DIV(FX,G,PHI,I,K,K1)
       P2=SECOND(T)

       P3=SECOND(T)
       WRITE(6,20) K
    20 FORMAT(*1DE EERSTE *,I3,* CYCLOTOMISCHE POLYNOMEN*//)
       DO 62 I=1,K
          J1=PHI(I,1)
          L=0
          DO 61 J=2,J1
             IF (PHI(I,J).EQ. 0) GO TO 61
                L=L+1
                COEF(L)=PHI(I,J)
                POW(L)=J-2
    61       CONTINUE
          WRITE(6,60) I,(COEF(J),POW(J),J=1,L)
    60    FORMAT(1H0,*PHI*,I3,3X,10(18(I2,1HX,I3,1H )/1H ,9X))
    62    CONTINUE
       RETURN
       END
```

```
      SUBROUTINE F(N,P,K1)
C X**N - 1
      INTEGER P(K1)
      P(1)=N+2
      P(2)=-1
      N1=N+1
      DO 10 I=3,N1
   10    P(I)=0
      P(N+2)=1
      RETURN
      END

      SUBROUTINE MUL(A,B,T,K,K1)
C A:=A*B[T, ]
      INTEGER A,B,T,C
      DIMENSION A(K1),B(K,K1),C(152)
      M=A(1)
      N=B(T,1)
      L=M+N-2
      A(1)=L
      DO 70 I=2,M
   70    C(I)=A(I)
      DO 80 I=2,L
   80    A(I)=0
      DO 90 I=2,M
         DO 90 J=2,N
   90       A(I+J-2)=A(I+J-2)+C(I)*B(T,J)
      RETURN
      END

      SUBROUTINE DIV(A,B,C,T,K,K1)
C C[T, ]:=A/B
      IMPLICIT INTEGER (A-Z)
      INTEGER A(K1),B(K1),C(K,K1)
      M=B(1)
  100 IF (B(M).NE.0) GO TO 110
         M=M-1
         GO TO 100
  110 J=A(1)-M+2
      A1=A(1)
      DO 120 I=2,J
  120    C(T,I)=0
      C(T,1)=J
      DO 130 N1=M,A1
         N=A1+M-N1
         IF (A(N).EQ.0) GO TO 130
            J=N-M+2
            IF (A(N)/B(M)*B(M).NE.A(N)) PRINT 140
  140          FORMAT(* DELING GAAT NIET OP*)
            KK=A(N)/B(M)
            DO 150 I=J,N
  150          A(I)=A(I)-KK*B(I-N+M)
            C(T,J)=KK
  130    CONTINUE
      RETURN
      END
```

```
PHI103   1X  0  1X  1  1X  2  1X  3  1X  4  1X  5  1X  6  1X  7  1X  8  1X  9  1X 10  1X 11  1X 12  1X 13  1X 14  1X 15  1X 16  1X 17
         1X 18  1X 19  1X 20  1X 21  1X 22  1X 23  1X 24  1X 25  1X 26  1X 27  1X 28  1X 29  1X 30  1X 31  1X 32  1X 33  1X 34  1X 35
         1X 36  1X 37  1X 38  1X 39  1X 40  1X 41  1X 42  1X 43  1X 44  1X 45  1X 46  1X 47  1X 48  1X 49  1X 50  1X 51  1X 52  1X 53
         1X 54  1X 55  1X 56  1X 57  1X 58  1X 59  1X 60  1X 61  1X 62  1X 63  1X 64  1X 65  1X 66  1X 67  1X 68  1X 69  1X 70  1X 71
         1X 72  1X 73  1X 74  1X 75  1X 76  1X 77  1X 78  1X 79  1X 80  1X 81  1X 82  1X 83  1X 84  1X 85  1X 86  1X 87  1X 88  1X 89
         1X 90  1X 91  1X 92  1X 93  1X 94  1X 95  1X 96  1X 97  1X 98  1X 99  1X100  1X101  1X102

PHI104   1X  0 -1X  4  1X  8 -1X 12  1X 16 -1X 20  1X 24 -1X 28  1X 32 -1X 36  1X 40 -1X 44  1X 48

PHI105   1X  0  1X  1  1X  2 -1X  5 -1X  6 -2X  7 -1X  8 -1X  9  1X 12  1X 13  1X 14  1X 15  1X 16  1X 17 -1X 20 -1X 22 -1X 24 -1X 26
        -1X 28  1X 31  1X 32  1X 33  1X 34  1X 35  1X 36 -1X 39 -1X 40 -2X 41 -1X 42 -1X 43  1X 46  1X 47  1X 48

PHI106   1X  0 -1X  1  1X  2 -1X  3  1X  4 -1X  5  1X  6 -1X  7  1X  8 -1X  9  1X 10 -1X 11  1X 12 -1X 13  1X 14 -1X 15  1X 16 -1X 17
         1X 18 -1X 19  1X 20 -1X 21  1X 22 -1X 23  1X 24 -1X 25  1X 26 -1X 27  1X 28 -1X 29  1X 30 -1X 31  1X 32 -1X 33  1X 34 -1X 35
         1X 36 -1X 37  1X 38 -1X 39  1X 40 -1X 41  1X 42 -1X 43  1X 44 -1X 45  1X 46 -1X 47  1X 48 -1X 49  1X 50 -1X 51  1X 52

PHI107   1X  0  1X  1  1X  2  1X  3  1X  4  1X  5  1X  6  1X  7  1X  8  1X  9  1X 10  1X 11  1X 12  1X 13  1X 14  1X 15  1X 16  1X 17
         1X 18  1X 19  1X 20  1X 21  1X 22  1X 23  1X 24  1X 25  1X 26  1X 27  1X 28  1X 29  1X 30  1X 31  1X 32  1X 33  1X 34  1X 35
         1X 36  1X 37  1X 38  1X 39  1X 40  1X 41  1X 42  1X 43  1X 44  1X 45  1X 46  1X 47  1X 48  1X 49  1X 50  1X 51  1X 52  1X 53
         1X 54  1X 55  1X 56  1X 57  1X 58  1X 59  1X 60  1X 61  1X 62  1X 63  1X 64  1X 65  1X 66  1X 67  1X 68  1X 69  1X 70  1X 71
         1X 72  1X 73  1X 74  1X 75  1X 76  1X 77  1X 78  1X 79  1X 80  1X 81  1X 82  1X 83  1X 84  1X 85  1X 86  1X 87  1X 88  1X 89
         1X 90  1X 91  1X 92  1X 93  1X 94  1X 95  1X 96  1X 97  1X 98  1X 99  1X100  1X101  1X102  1X103  1X104  1X105  1X106

PHI108   1X  0 -1X 18  1X 36
```

9.3. Input/output

It is frequently necessary for a program to perform input/output, and quite often character by character. We have therefore run a number of programs to do such operations with various variations.

Several programming languages do not provide true character input/output and one is instead forced to read an entire line at a time and write extra code to pick it apart.

The test performed was to copy the first 200 lines of the file A68DOC to the OUTPUT file. A68DOC contained the documentation for the CDC Algol 68 compiler; it is a character file with varyinglength lines. If a language is unable to represent varying length lines faithfully to the programmer and must instead pad all lines to some fixed length (such as 80 characters), then the extra CPU time wasted will be charged against is, and not against another language which avoids processing nonexistent characters.

In each language, language features were found or procedures written to read and write single characters, one character in or out for each call.

In Pascal, single character input-output is provided directly by the language. It was therefore used directly in the run labelled "c". In the run "cs", Pascal's string output was used. Pascal does not provide string input on character files.

In Algol 68, single character input/output is provided by the language, but is absurdly slow. There appears to be a large fixed overhead associated with each input/output operation that is independent of the amount of input or output to be performed. Reading or printing an entire line takes only about twice as much time as reading or printing a single character. Three tests were therefore performed on Algol 68,

"c" - use the language's character i/o.

"s" - use the language's string i/o, reading and writing entire lines at a time.

"scs" - write character i/o procedures which use the string i/o internally, and then use these procedures to copy the file.

In Fortran, no character i/o is provided by the language. An input operation always processes an entire line, and if the entire line is not

completely read, the rest of it is simply lost. Therefore, it was decided to read 80 characters from each line, in the hope that this would be sufficient. The fact that it was necessary to hope already shows a deficiency in Fortran. It turned out that A68DOC indeed did contain a few lines that were longer than 80 characters, but these were, fortunately, not among the first 200. The following tests were performed on Fortran. In each case, entire lines were read and written; and no character-at-a-time procedures were used:

"A1" - the format 80A1 was used with formatted i/o.

"A10" - 8A10 was used with formatted i/o.

In Algol 60, we again used various methods.

"char" - the character i/o routines "in character" and "out character" were used.

"8A" - the format "8A" was used to read and write blocks of 8 characters at a time.

"A" - the "A" format was used to read and write single characters.

It was felt that the presence or absence of array-bound checking would make little difference to the measured results, since in these programs

(1) little array bound checking needs to be done, and

(2) Extended Fortran refuses to perform array-bound checking on input/output operations even if asked to.

The execution times are as follows:

			CP time per line (milliseconds)
Pascal	ch	c	2.8
		cs	7.3
Algol 68		c	245
		scs	100
		s	50
Fortran		A1	27.5
		A10	5.0
MNF		A1	25.5
		A10	5.0

Algol 3	pch	char	115
		A	260
		8A	40
Algol 4	⌐ch	char	205
		A	330
		8A	55

It is possible that experienced users of each language may have found sneaky ways to reduce character i/o times; however, a casual user is about as likely to strike upon such special techniques as the authors were.

9.4. Feature timings

We also performed a number of runs in order to determine how efficiently various classes of language features are implemented. To do this for each language feature, a loop of the form

```
for i := 1 until 10000 do test statement;
```

was timed. From this the time taken by an empty loop

```
for i := 1 until 10000 do;
```

is subtracted, and the result divided by the number of iterations. The results appear in the accompanying tables.

WICHMANN [15] has obtained, by actual measurement, the frequencies in which Algol 60 features are used during Algol 60 program execution in real life. These frequencies have been used here to compute weighted averages of the various feature timings. These averages should not be taken too seriously, though, because

- Patterns of usage in different programming languages are likely to differ; something often used in one language may be used hardly at all in another.
- The tests themselves differ slightly from those performed by Wichmann.
- Loop optimization in a compiler can cause the feature timings to go awry.

In any case, to determine what was actually timed, the program listings should be examined. The Pascal and Fortran timings were run with loops

of 50,000 iterations, whereas the Algol 60 and Algol 68 timings were done with runs of 10,000 iterations. Timings were performed on a CDC Cyber 73 (functionally equivalent to a twin CPU CDC 6400 or a CDC 6500) under Scope 3.4.1. An attempt was made to repeat some of the timings under Scope 3.4.4, but the CPU interval timer had become sufficiently irregular on the newer system that this attempt had to be abandoned. It is reasonable to suppose, however, that the CPU itself has not changed in speed with the change to a new version of the operating system. Some of the Algol 68 timings were made under Scope 3.4.4 with the old system's clock, which provided reasonable precision.

All times are given in microseconds. The digit after the decimal point should not be considered significant, but is provided in case the reader wishes to use statistical noise reduction techniques. The precision of the figures can be judged by examining them for internal consistency. They reflect a compromise between the costs of performing tests and the imprecision of the clock.

9.4.1. Pascal timings

ch	¬ch	
3.2	3.0	X:=1.1
6.9	3.8	X:=11
3.2	2.9	X:=Y
4.6	4.5	X:=Y+Z
8.9	8.7	X:=Y*Z
9.1	8.6	X:=Y/Z
1.7	2.0	K:=11
9.4	9.5	K:=ROUND(1.1)
3.6	3.6	K:=L+M
11.4	10.3	K:=L*M
19.5	14.3	K:=L DIV M
3.7	3.6	K:=L
6.9	4.2	X:=L
9.3	10.4	L:=ROUND(Y)
9.0	8.8	X:=SQR(Y)
14.0	14.7	X:=SQR(Y)*Y
251.1	249.5	X:=EXP(Z*LN(Y))
9.1	5.1	E[J]:=L
14.6	8.6	E2[J,F]:=L
20.1	9.2	E3[J,F,G]:=L
9.9	5.7	L:=E[J]
18.8	18.2	VAR A:REAL;BEGIN A:=3.14 END
19.1	18.4	VAR A:ARRAY[1..1] OF REAL;BEGIN A[1]:=2.72 END
19.2	18.3	VAR A:ARRAY[1..500] OF REAL;BEGIN A[23]:=7.8 END
19.7	18.3	VAR A:ARRAY[1..1,1..1] OF REAL;BEGIN A[1,1]:=4.65 END
18.8	18.1	VAR A:ARRAY[1..1,1..1,1..1] OF REAL;BEGIN A[1,1,1]:=1.7 END
17.3	17.2	LABEL 1;BEGIN GOTO 1;1: END
13.4	7.5	CASE J OF 1: END
134.1	129.9	X:=SIN(Y)
129.2	127.1	X:=COS(Y)
5.2	3.9	X:=ABS(Y)
131.3	128.5	X:=EXP(Y)
126.7	126.5	X:=LN(Y)
93.3	93.2	X:=SQRT(Y)
134.7	133.2	X:=ARCTAN(Y)
16.8	11.5	IF Y>0 THEN X:=1 ELSE IF Y=0 THEN X:=0 ELSE X:=-1
10.7	6.7	X:=TRUNC(Y)
18.7	18.7	P0
23.5	22.9	P1(X)
27.1	27.2	P2(X,Y)
28.7	28.9	P3(X,Y,Z)
6.8	6.8	LOOP OF FOR I:=1 TO N DO;
12.5	10.9	MIXTURE

ch: range checking is performed.

¬ch: no range checking is performed.

```
PROGRAM TIMER(OUTPUT);
(* TIMER   (PASCAL EXECUTION TIME,IDEA WICHMANN)*)

CONST N=50000;M=11;Y=1.1;Z=1.1;

VAR I,J,F,G,EP,K,L,T:INTEGER;P,Q,C,FSUM,S,RELFREQ,LOAD,TYD,X:REAL;
    TT,FREQ:ARRAY[1..42] OF REAL;E:ARRAY[1..1] OF INTEGER;
    E2:ARRAY[1..1,1..1] OF INTEGER;E3:ARRAY[1..1,1..1,1..1] OF INTEGER;

PROCEDURE SAVE;BEGIN EP:=EP+1;TT[EP]:=(Q-P-C)/N END;

PROCEDURE NOTE;
          BEGIN EP:=EP+1;RELFREQ:=FREQ[EP]*FSUM;TYD:=1E3*TT[EP];
                LOAD:=RELFREQ*TYD;S:=S+LOAD;
                WRITELN;
                WRITE(' ',RELFREQ:5:3,LOAD:6:1,TYD:7:1,'  ');
          END;

PROCEDURE P0;BEGIN X:=3.14 END;
PROCEDURE P1(X:REAL);BEGIN X:=2.71 END;
PROCEDURE P2(X,Y:REAL);BEGIN X:=1.25 END;
PROCEDURE P3(X,Y,Z:REAL);BEGIN Z:=5.6 END;
PROCEDURE Q1;VAR A:REAL;BEGIN A:=3.14 END;
PROCEDURE Q2;VAR A:ARRAY[1..1] OF REAL;BEGIN A[1]:=2.72 END;
PROCEDURE Q3;VAR A:ARRAY[1..500] OF REAL;BEGIN A[23]:=7.8 END;
PROCEDURE Q4;VAR A:ARRAY[1..1,1..1] OF REAL;BEGIN A[1,1]:=4.65 END;
PROCEDURE Q5;VAR A:ARRAY[1..1,1..1,1..1] OF REAL;
             BEGIN A[1,1,1]:=1.7 END;
PROCEDURE Q6;LABEL 1;BEGIN GOTO 1;1: END;

PROCEDURE EXTRA1;
BEGIN
  P:=CLOCK;FOR I:=1 TO N DO X:=1.1;
     Q:=CLOCK;SAVE;
  P:=CLOCK;FOR I:=1 TO N DO X:=11;
     Q:=CLOCK;SAVE;
  P:=CLOCK;FOR I:=1 TO N DO X:=Y;
     Q:=CLOCK;SAVE;
  P:=CLOCK;FOR I:=1 TO N DO X:=Y+Z;
     Q:=CLOCK;SAVE;
  P:=CLOCK;FOR I:=1 TO N DO X:=Y*Z;
     Q:=CLOCK;SAVE;
  P:=CLOCK;FOR I:=1 TO N DO X:=Y/Z;
     Q:=CLOCK;SAVE;
  P:=CLOCK;FOR I:=1 TO N DO K:=11;
     Q:=CLOCK;SAVE;
  P:=CLOCK;FOR I:=1 TO N DO K:=ROUND(1.1);
     Q:=CLOCK;SAVE;
  P:=CLOCK;FOR I:=1 TO N DO K:=L+M;
     Q:=CLOCK;SAVE;
  P:=CLOCK;FOR I:=1 TO N DO K:=L*M;
     Q:=CLOCK;SAVE;

  P:=CLOCK;FOR I:=1 TO N DO K:=L DIV M;
     Q:=CLOCK;SAVE;
  P:=CLOCK;FOR I:=1 TO N DO K:=L;
     Q:=CLOCK;SAVE;
  P:=CLOCK;FOR I:=1 TO N DO X:=L;
     Q:=CLOCK;SAVE;
```

```
P:=CLOCK;FOR I:=1 TO N DO L:=ROUND(Y);
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO X:=SQR(Y);
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO X:=SQR(Y)*Y;
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO X:=EXP(Z*LN(Y));
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO E[J]:=L;
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO E2[J,F]:=L;
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO E3[J,F,G]:=L;
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO L:=E[J];
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO Q1;
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO Q2;
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO Q3;
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO Q4;
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO Q5;
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO Q6;
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO CASE J OF 1:  END;
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO X:=SIN(Y);
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO X:=COS(Y);
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO X:=ABS(Y);
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO X:=EXP(Y);
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO X:=LN(Y);
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO X:=SQRT(Y);
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO X:=ARCTAN(Y);
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO IF Y>0 THEN X:=1
                            ELSE IF Y=0 THEN X:=0
                                        ELSE X:=-1;

   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO X:=TRUNC(Y);
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO P0;
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO P1(X);
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO P2(X,Y);
   Q:=CLOCK;SAVE;
P:=CLOCK;FOR I:=1 TO N DO P3(X,Y,Z);
   Q:=CLOCK;SAVE;
END;
```

```
PROCEDURE EXTRA2;
BEGIN
  NOTE;WRITE('X:=1.1');
  NOTE;WRITE('X:=11');
  NOTE;WRITE('X:=Y');
  NOTE;WRITE('X:=Y+Z');
  NOTE;WRITE('X:=Y*Z');
  NOTE;WRITE('X:=Y/Z');
  NOTE;WRITE('K:=11');
  NOTE;WRITE('K:=ROUND(1.1)');
  NOTE;WRITE('K:=L+M');
  NOTE;WRITE('K:=L*M');
  NOTE;WRITE('K:=L DIV M');
  NOTE;WRITE('K:=L');
  NOTE;WRITE('X:=L');
  NOTE;WRITE('L:=ROUND(Y)');
  NOTE;WRITE('X:=SQR(Y)');
  NOTE;WRITE('X:=SQR(Y)*Y');
  NOTE;WRITE('X:=EXP(Z*LN(Y))');
  NOTE;WRITE('E[J]:=L');
  NOTE;WRITE('E2[J,F]:=L');
  NOTE;WRITE('E3[J,F,G]:=L');
  NOTE;WRITE('L:=E[J]');
  NOTE;WRITE('VAR A:REAL;BEGIN A:=3.14 END');
  NOTE;WRITE('VAR A:ARRAY[1..1] OF REAL;BEGIN A[1]:=2.72 END');
  NOTE;WRITE('VAR A:ARRAY[1..500] OF REAL;BEGIN A[23]:=7.8 END');
  NOTE;WRITE('VAR A:ARRAY[1..1,1..1] OF REAL;BEGIN A[1,1]:=4.65 END');
  NOTE;WRITELN('VAR A:ARRAY[1..1,1..1,1..1] OF REAL;');
       WRITE('                                BEGIN A[1,1,1]:=1.7 END');
  NOTE;WRITE('LABEL 1;BEGIN GOTO 1;1: END');
  NOTE;WRITE('CASE J OF 1: END');
  NOTE;WRITE('X:=SIN(Y)');
  NOTE;WRITE('X:=COS(Y)');
  NOTE;WRITE('X:=ABS(Y)');
  NOTE;WRITE('X:=EXP(Y)');
  NOTE;WRITE('X:=LN(Y)');
  NOTE;WRITE('X:=SQRT(Y)');
  NOTE;WRITE('X:=ARCTAN(Y)');

  NOTE;WRITE('IF Y>0 THEN X:=1 ELSE IF Y=0 THEN X:=0 ELSE X:=-1');
  NOTE;WRITE('X:=TRUNC(Y)');
  NOTE;WRITE('P0');
  NOTE;WRITE('P1(X)');
  NOTE;WRITE('P2(X,Y)');
  NOTE;WRITE('P3(X,Y,Z)');
  NOTE;WRITE('LOOP OF FOR I:=1 TO N DO ;');
END;
```

```
BEGIN
L:=11;E[1]:=11;FSUM:=0;

FOR I:=1 TO 42 DO
  CASE I OF
  1,3:FREQ[I]:=10000;
    2:FREQ[I]:=7000;
    4:FREQ[I]:=26682;
    5:FREQ[I]:=31210;
    6:FREQ[I]:=11000;
    7:FREQ[I]:=3000;
 8,14:FREQ[I]:=500;
    9:FREQ[I]:=4300;
   10:FREQ[I]:=4978;
   11:FREQ[I]:=480.9;
   12:FREQ[I]:=5000;
   13:FREQ[I]:=4000;
   15:FREQ[I]:=2780;
   16:FREQ[I]:=309;
   17:FREQ[I]:=442;
18,21:FREQ[I]:=23795;
   19:FREQ[I]:=15963;
   20:FREQ[I]:=296;
   22:FREQ[I]:=0.0;
23,24:FREQ[I]:=59;
   25:FREQ[I]:=39;
   26:FREQ[I]:=0.73;
   27:FREQ[I]:=2010;
   28:FREQ[I]:=94;
   29:FREQ[I]:=1020;
   30:FREQ[I]:=1490;
   31:FREQ[I]:=1390;
   32:FREQ[I]:=831;
   33:FREQ[I]:=644;
   34:FREQ[I]:=1750;
   35:FREQ[I]:=591;
   36:FREQ[I]:=81.9;
   37:FREQ[I]:=909;
   38:FREQ[I]:=788;
39,40:FREQ[I]:=2316;
   41:FREQ[I]:=6053;
   42:FREQ[I]:=17800;
END;

FOR I:=1 TO 42 DO FSUM:=FSUM+FREQ[I];
FSUM:=42/FSUM;

C:=0;FOR EP:=1 TO 10 DO
BEGIN P:=CLOCK;FOR I:=1 TO N DO ;
      Q:=CLOCK;C:=C+(Q-P)
END;

C:=C/10;TT[42]:=C/N;EP:=0;J:=1;F:=1;G:=1;

EXTRA1;
WRITELN;
WRITELN('        PASCAL EXECUTION TIME');
WRITELN;
WRITELN('  FREQ    WEIGHT  TIME STATEMENT');
EP:=0;S:=0;
EXTRA2;
S:=S/42;
WRITELN;WRITELN;
WRITELN('  1.000',S:6:1,S:7:1,'  MIXTURE');
END.
```

9.4.2 Algol 68 timings

ch	ch,Z	A	A,Z	
4.9	3.6	4.8	4.5	X:=1.1
5.5	4.3	6.2	4.8	X:=11
6.1	6.2	6.6	5.6	X:=Y
10.0	9.5	11.7	10.2	X:=Y+Z
14.8	12.6	14.0	13.7	X:=Y*Z
13.4	14.1	13.5	14.1	X:=Y/Z
4.6	5.8	3.5	4.1	K:=11
8.9	10.3	8.9	8.7	K:= ROUND 1.1
8.0	7.6	7.3	7.7	K:=L+M
14.3	14.1	14.9	13.6	K:=L*M
18.1	18.5	19.1	17.4	K:=L OVER M
6.5	5.4	5.8	6.0	K:=L
6.6	6.7	6.5	6.0	X:=L
10.8	10.8	10.6	10.1	L:= ROUND Y
35.8	35.5	34.6	36.4	X:=Y**2
39.9	41.1	40.1	41.4	X:=Y**3
301.7	305.4	302.5	294.3	X:=EXP(Z*LN(Y))
53.3	50.2	23.7	23.5	E[J]:=L
76.9	74.5	39.8	37.6	E2[J,F]:=L
101.9	100.6	51.9	51.9	E3[J,F,G]:=L
58.5	56.7	25.2	23.7	L:=E[J]
22.6	20.8	21.4	19.4	REAL A:=3.14;A
220.0	224.6	220.2	215.1	[1:1] REAL A:=(2.72);A
1303.5	1386.7	1365.9	1288.2	[1:500] REAL A;A[23]:=7.8
258.9	269.4	273.9	256.5	[1:1,1:1] REAL A:=((4.65));A
305.7	316.0	316.8	312.4	[1:1,1:1,1:1] REAL A:=(((1.7)));A
2.0	0.5	1.2	0.8	ABCD;ABCD: SKIP
70.4	73.9	73.0	69.2	PROC SS= VOID :PQ;SS;PQ: SKIP
166.5	170.3	173.9	171.1	X:=SIN(Y)
159.9	167.7	169.8	168.7	X:=COS(Y)
6.6	6.1	7.6	6.4	X:=ABS Y
128.8	135.8	134.8	133.7	X:=EXP(Y)
165.7	165.3	167.1	163.2	X:=LN(Y)
113.4	110.6	113.7	109.1	X:=SORT(Y)
166.1	171.0	170.7	177.2	X:=ARCTAN(Y)
7.5	7.1	7.9	7.6	X:=SIGN Y
13.7	14.7	15.1	14.3	X:=ENTIER Y
52.4	51.4	52.9	52.4	P0
46.8	45.6	47.8	45.0	P1(X)
50.3	49.1	52.1	48.4	P2(X,Y)
54.5	51.9	54.7	51.4	P3(X,Y,Z)
6.5	7.1	6.5	6.8	LOOP OF TO N DO
31.6	30.5	22.6	21.7	MIXTURE

ch: Array subscript checking is performed

A: No array subscript checking is performed

Z: some extra object code optimization is performed (apparently with little effect).

```
     #TIMER    (ALGOL68 EXECUTION TIME,IDEA WICHMANN) #
'BEGIN'
   'INT' I,J:=1,F:=1,G:=1,EP,'INT' N=50000;
   'REAL' P,Q,C,FSUM,S,RELFREQ,LOAD,TIME;
   [1:42] 'REAL' TT,FREQ;

   'PROC' SAVE='VOID':(EP+:=1;TT[EP]:=(Q-P-C)/N);

   'PROC'  NOTE=([ ]'CHAR' STAT) 'VOID':
                (EP+:=1;RELFREQ:=FREQ[EP]*FSUM;TIME:=1E6*TT[EP];
                 LOAD:=RELFREQ*TIME;S:=S+LOAD;
                 PRINT((NEWLINE,FIXED(RELFREQ,6,3),
                        FIXED(LOAD,7,1),FIXED(TIME,8,1),"   ",STAT))
                );

   'INT' K,L,'INT' M=11;
   'REAL' X,Y:=1.1,Z:=1.1;
   [1:1] 'INT' E,[1:1,1:1] 'INT' E2,[1:1,1:1,1:1] 'INT' E3;

   'PROC' P0='VOID':X:=3.14;
   'PROC' P1=('REF' 'REAL' X) 'VOID':X:=1.25;
   'PROC' P2=('REF' 'REAL' X,Y) 'VOID':Y:=1.4;
   'PROC' P3=('REF' 'REAL' X,Y,Z) 'VOID':Z:=5.6;

   L:=E[1]:=11;FSUM:=0;

   'PROC' XX=('INT' N) 'REAL':
             'CASE' N 'IN' 10000, 7000,10000,26682,31210,
                           11000,3000,500,4300,4978,480.9,5000,
                           4000,500,2780,309,442,23795,15963,296,
                           23795,0,59,59,39,.73,2010,94,1020,1490,
                           1390,831,644,1750,591,81.9,909,788,
                           2316,2316,6053,17800
             'ESAC';

   'FOR' I 'TO' 42 'DO' FREQ[I]:=XX(I);FSUM:=FSUM+FREQ[I] 'OD';

   FSUM:=42/FSUM;
   PRINT((NEWLINE,"ALGOL68 EXECUTION TIME",NEWLINE,NEWLINE,
          " FREQ  WEIGHT  TIME    STATEMENT",NEWLINE));

   C:=0;'TO' 10
        'DO' P:=CLOCK;'TO' N 'DO' 'SKIP' 'OD';
             Q:=CLOCK;C:=C+(Q-P)
        'OD';

   C/:=10;TT[42]:=C/N;EP:=0;

   P:=CLOCK;'TO' N 'DO' X:=1.1 'OD';
   Q:=CLOCK;SAVE;
   P:=CLOCK;'TO' N 'DO' X:=11 'OD';
         Q:=CLOCK;SAVE;
   P:=CLOCK;'TO' N 'DO' X:=Y 'OD';
         Q:=CLOCK;SAVE;
   P:=CLOCK;'TO' N 'DO' X:=Y+Z 'OD';
         Q:=CLOCK;SAVE;
   P:=CLOCK;'TO' N 'DO' X:=Y*Z 'OD';
         Q:=CLOCK;SAVE;
   P:=CLOCK;'TO' N 'DO' X:=Y/Z 'OD';
         Q:=CLOCK;SAVE;
   P:=CLOCK;'TO' N 'DO' K:=11 'OD';
         Q:=CLOCK;SAVE;
   P:=CLOCK;'TO' N 'DO' K:='ROUND' 1.1 'OD';
         Q:=CLOCK;SAVE;
```

```
P:=CLOCK;'TO' N 'DO' K:=L+M 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' K:=L*M 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' K:=L 'OVER' M 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' K:=L 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' X:=L 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' L:='ROUND' Y 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' X:=Y**2 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' X:=Y**3 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' X:=EXP(Z*LN(Y)) 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' E[J]:=L 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' E2[J,F]:=L 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' E3[J,F,G]:=L 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' L:=E[J] 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' 'REAL' A:=3.14;A 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' [1:1] 'REAL' A:=(2.72);A 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' [1:500] 'REAL' A;A[23]:=7.8 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' [1:1,1:1] 'REAL' A:=((4.65));A 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' [1:1,1:1,1:1] 'REAL' A:=(((1.7)));A 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' ABCD;ABCD: 'SKIP' 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' 'PROC' SS='VOID':PQ;SS;PQ: 'SKIP' 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' X:=SIN(Y) 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' X:=COS(Y) 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' X:='ABS' Y 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' X:=EXP(Y) 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' X:=LN(Y) 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' X:=SQRT(Y) 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' X:=ARCTAN(Y) 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' X:='SIGN' Y 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' X:='ENTIER' Y 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' P0 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' P1(X) 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' P2(X,Y) 'OD';
      Q:=CLOCK;SAVE;
P:=CLOCK;'TO' N 'DO' P3(X,Y,Z) 'OD';
      Q:=CLOCK;SAVE;

EP:=0;S:=0;
```

```
    NOTE("X:=1.1");
    NOTE("X:=11");
    NOTE("X:=Y");
    NOTE("X:=Y+Z");
    NOTE("X:=Y*Z");
    NOTE("X:=Y/Z");
    NOTE("K:=11");
    NOTE("K:='ROUND' 1.1");
    NOTE("K:=L+M");
    NOTE("K:=L*M");
    NOTE("K:=L 'OVER' M");
    NOTE("K:=L");
    NOTE("X:=L");
    NOTE("L:='ROUND' Y");
    NOTE("X:=Y**2");
    NOTE("X:=Y**3");
    NOTE("X:=EXP(Z*LN(Y))");
    NOTE("E[J]:=L ");
    NOTE("E2[J,F]:=L ");
    NOTE("E3[J,F,G]:=L ");
    NOTE("L:=E[J]");
    NOTE(" 'REAL' A:=3.14;A");
    NOTE("[1:1] 'REAL' A:=(2.72);A");
    NOTE("[1:500] 'REAL' A;A[23]:=7.8 ");
    NOTE("[1:1,1:1] 'REAL' A:=((4.65));A");
    NOTE("[1:1,1:1,1:1] 'REAL' A:=(((1.7)));A");
    NOTE("ABCD;ABCD: 'SKIP'");
    NOTE(" 'PROC' SS='VOID':PQ;SS;PQ: 'SKIP'");
    NOTE("X:=SIN(Y)");
    NOTE("X:=COS(Y)");
    NOTE("X:=ABS Y");
    NOTE("X:=EXP(Y)");
    NOTE("X:=LN(Y)");
    NOTE("X:=SQRT(Y)");
    NOTE("X:=ARCTAN(Y)");
    NOTE("X:=SIGN Y ");
    NOTE("X:=ENTIER Y ");
    NOTE("P0");
    NOTE("P1(X)");
    NOTE("P2(X,Y)");
    NOTE("P3(X,Y,Z)");
    NOTE("LOOP OF 'TO' N 'DO' ");

S/:=42;
PRINT((NEWLINE,NEWLINE,"+1     ",FIXED(S,7,1),FIXED(S,8,1),
       "  MIXTURE",NEWLINE))

'END'
```

9.4.3.1 Fortran Extended timings

D,OPT=0 pch	OPT=0 ⌐lch	OPT=1 ⌐lch	OPT=2 ⌐lch	
3.9	4.0	3.3	1.0	X=1.1
2.9	3.0	3.2	3.5	X=11
2.9	3.5	3.0	1.1	X=Y
6.0	5.9	5.8	1.2	X=Y+Z
10.2	10.6	9.5	1.2	X=Y*Z
10.2	10.3	9.5	1.3	X=Y/Z
1.9	3.1	1.7	2.1	K=11
3.8	3.7	3.7	2.9	K=1.1
5.1	5.9	4.5	1.4	K=L+M
10.3	10.6	9.8	1.3	K=L*M
13.9	13.7	13.0	1.4	K=L/M
2.9	3.8	3.2	1.9	K=L
3.8	3.7	3.6	2.7	X=L
3.9	4.0	4.1	3.2	L=Y
9.1	8.1	8.2	6.3	K=Y**2
13.8	14.3	14.2	13.3	X=Y**3
233.9	235.0	222.8	223.4	X=Y**Z
40.6	5.5	3.2	1.1	E(J)=L
50.4	7.7	3.1	1.3	E2(J,F)=L
67.0	8.9	3.3	1.2	E3(J,F,G)=L
41.2	5.0	3.3	3.2	L=E(J)
8.5	8.6	13.3	13.9	SUBROUTINE A=3.14 END
9.0	8.6	13.9	13.9	SUBROUTINE REAL A(1) A(1)=2.72 END
8.9	8.6	14.4	13.6	SUBR. REAL A(500) A(23)=7.8 END
8.2	8.6	13.5	13.8	SUBR. REAL A(1,1) A(1,1)=4.65 END
9.0	8.4	14.0	13.6	SUBR. REAL A(1,1,1) A(1,1,1)= 1.7 END
5.7	4.9	9.5	10.0	SUBROUTINE GOTO 26 26 END
8.2	7.7	11.0	10.5	GOTO (127,227),J 127 CONTINUE
135.7	136.4	133.4	131.2	X=SIN(Y)
132.7	131.6	126.9	126.9	X=COS(Y)
10.3	10.5	3.8	2.6	X=ABS(Y)
98.4	98.0	89.7	89.6	X=EXP(Y)
129.8	128.7	125.2	125.6	X=ALOG(Y)
77.1	77.3	72.7	72.6	X=SQRT(Y)
134.6	135.7	131.1	131.8	X=ATAN(Y)
15.3	15.2	12.2	12.0	X=0 IF(Y.GT.0) X=1 IF (Y.LT.0) X=-1
13.0	12.6	4.8	3.7	X=INT(Y)
8.7	8.5	13.8	13.7	CALL P0
36.0	36.9	21.7	22.7	CALL P1(X)
53.3	52.7	21.6	21.7	CALL P2(X,Y)
69.8	69.9	21.4	21.9	CALL P3(X,Y,Z)
7.4	7.5	2.5	2.5	LOOP OF DO 40 I=1,N 40 CONTINUE
23.1	12.7	9.2	6.3	MIXTURE

¬ch: no subscript checking

pch: partial checking

The figures given above for OPT=2 connot be trusted, since the optimizer may well be capable of removing part of the calculation being measured out of the loop. The suspiciously low timings for some statements suggest that this may have happened. CDC provides their own measurements of the Fortran mathematics routines in [19].

```
      PROGRAM TIMER(OUTPUT,TAPE6=OUTPUT)
C TIMER  (FORTRAN EXECUTION TIME , IDEA WICHMANN)
      COMMON/L1/EP,TT(42)/L2/Q,P,C,N/L3/RELFREQ,FREQ(42),FSUM,LOAD,TIJD,
     CS,B1,B2,B3,B4/L4/X
      INTEGER I,J,F,G,N,EP,K,L,M,E(1),E2(1,1),E3(1,1,1),B1,B2,B3,B4
      REAL P,Q,C,FSUM,S,RELFREQ,LOAD,TIJD,X,Y,Z,T,TT,FREQ
      DATA FREQ/10000.,7000.,10000.,26682.,31210.,11000.,3000.,500.,
     C4300.,4978.,480.9,5000.,4000.,500.,2780.,309.,442.,23795.,15963.,
     C296.,23795.,0.,59.,59.,39.,.73,2010.,94.,1020.,1490.,1390.,831.,
     C644.,1750.,591.,81.9,909.,788.,2316.,2316.,6053.,17800./
      N=50000 $M=11 $Y=1.1 $Z=1.1
      L=11 $E(1)=11 $FSUM=0.
      J=1 $F=1 $G=1
      DO 10 I=1,42
   10 FSUM=FSUM+FREQ(I)
      FSUM=42./FSUM
      PRINT 15
   15 FORMAT(8X,*FORTRAN EXECUTION TIME  (50000 KEER) *//)
      PRINT 16
   16 FORMAT(*   FREQ  WEIGHT   TIME  STATEMENT *,/)
      C=0.
      DO 20 EP=1,10
      P=SECOND(T)
      DO 30 I=1,N
   30 CONTINUE
      Q=SECOND(T)
   20 C=Q-P+C
      C=C/10. $TT(42)=C/N $EP=0
      P=SECOND(T)
      DO 100 I=1,N
  100 X=1.1
      Q=SECOND(T) $CALL SAVE
      P=SECOND(T)
      DO 101 I=1,N
  101 X=11
      Q=SECOND(T) $CALL SAVE
      P=SECOND(T)
      DO 102 I=1,N
  102 X=Y
      Q=SECOND(T) $CALL SAVE
      P=SECOND(T)
      DO 103 I=1,N
  103 X=Y+Z
      Q=SECOND(T) $CALL SAVE
      P=SECOND(T)
      DO 104 I=1,N
  104 X=Y*Z
      Q=SECOND(T) $CALL SAVE
      P=SECOND(T)
      DO 105 I=1,N
  105 X=Y/Z
      Q=SECOND(T) $CALL SAVE
      P=SECOND(T)
      DO 106 I=1,N
  106 K=11
      Q=SECOND(T) $CALL SAVE
      P=SECOND(T)
      DO 107 I=1,N
  107 K=1.1
      Q=SECOND(T) $CALL SAVE
      P=SECOND(T)
      DO 108 I=1,N
```

```
  108 K=L+M
      Q=SECOND(T) $CALL SAVE
      P=SECOND(T)
      DO 109 I=1,N
  109 K=L*M
      Q=SECOND(T) $CALL SAVE
      P=SECOND(T)
      DO 110 I=1,N
  110 K=L/M
      Q=SECOND(T) $CALL SAVE
      P=SECOND(T)
      DO 111 I=1,N
  111 K=L
      Q=SECOND(T) $CALL SAVE
      P=SECOND(T)
      DO 112 I=1,N
  112 X=L
      Q=SECOND(T) $CALL SAVE
      P=SECOND(T)
      DO 113 I=1,N
  113 L=Y
      Q=SECOND(T) $CALL SAVE
      P=SECOND(T)
      DO 114 I=1,N
  114 X=Y**2
      Q=SECOND(T) $CALL SAVE
      P=SECOND(T)
      DO 115 I=1,N
  115 X=Y**3
      Q=SECOND(T) $CALL SAVE
      P=SECOND(T)
      DO 116 I=1,N
  116 X=Y**Z
      Q=SECOND(T) $CALL SAVE
C$    TRACE SUBSCRIPTS
      P=SECOND(T)
      DO 117 I=1,N
  117 E(J)=L
      Q=SECOND(T) $CALL SAVE
      P=SECOND(T)
      DO 118 I=1,N
  118 E2(J,F)=L
      Q=SECOND(T) $CALL SAVE
      P=SECOND(T)
      DO 119 I=1,N
  119 E3(J,F,G)=L
      Q=SECOND(T) $CALL SAVE
      P=SECOND(T)
      DO 120 I=1,N
  120 L=E(J)
      Q=SECOND(T) $CALL SAVE
C$    NO TRACE SUBSCRIPTS
      P=SECOND(T)
      DO 121 I=1,N
  121 CALL Q1
      Q=SECOND(T) $CALL SAVE
      P=SECOND(T)
      DO 122 I=1,N
  122 CALL Q2
      Q=SECOND(T) $CALL SAVE
      P=SECOND(T)
      DO 123 I=1,N
```

```
123 CALL Q3
    Q=SECOND(T) $CALL SAVE
    P=SECOND(T)
    DO 124 I=1,N
124 CALL Q4
    Q=SECOND(T) $CALL SAVE
    P=SECOND(T)
    DO 125 I=1,N
125 CALL Q5
    Q=SECOND(T) $CALL SAVE
    P=SECOND(T)
    DO 126 I=1,N
126 CALL Q6
    Q=SECOND(T) $CALL SAVE
    P=SECOND(T)
    DO 127 I=1,N
    GOTO (127,227),J
127 CONTINUE
227 Q=SECOND(T) $CALL SAVE
    P=SECOND(T)
    DO 128 I=1,N
128 X=SIN(Y)
    Q=SECOND(T) $CALL SAVE
    P=SECOND(T)
    DO 129 I=1,N
129 X=COS(Y)
    Q=SECOND(T) $CALL SAVE
    P=SECOND(T)
    DO 130 I=1,N
130 X=ABS(Y)
    Q=SECOND(T) $CALL SAVE
    P=SECOND(T)
    DO 131 I=1,N
131 X=EXP(Y)
    Q=SECOND(T) $CALL SAVE
    P=SECOND(T)
    DO 132 I=1,N
132 X=ALOG(Y)
    Q=SECOND(T) $CALL SAVE
    P=SECOND(T)
    DO 133 I=1,N
133 X=SQRT(Y)
    Q=SECOND(T) $CALL SAVE
    P=SECOND(T)
    DO 134 I=1,N
134 X=ATAN(Y)
    Q=SECOND(T) $CALL SAVE
    P=SECOND(T)
    DO 135 I=1,N
    X=0
    IF (Y.GT.0) X=1
135 IF (Y.LT.0) X=-1
    Q=SECOND(T) $CALL SAVE
    P=SECOND(T)
    DO 136 I=1,N
136 X=INT(Y)
    Q=SECOND(T) $CALL SAVE
    P=SECOND(T)
    DO 137 I=1,N
137 CALL P0
    Q=SECOND(T) $CALL SAVE
    P=SECOND(T)
    DO 138 I=1,N
```

```
138 CALL P1(X)
    Q=SECOND(T) $CALL SAVE
    P=SECOND(T)
    DO 139 I=1,N
139 CALL P2(X,Y)
    Q=SECOND(T) $CALL SAVE
    P=SECOND(T)
    DO 140 I=1,N
140 CALL P3(X,Y,Z)
    Q=SECOND(T) $CALL SAVE
    EP=0.  $S=0.
    B1="X=1.1" $B2=" " $B3=" " $B4=" " $CALL NOTE
    B1="X=11" $CALL NOTE
    B1="X=Y" $CALL NOTE
    B1="X=Y+Z" $CALL NOTE
    B1="X=Y*Z" $CALL NOTE
    B1="X=Y/Z" $CALL NOTE
    B1="K=11" $CALL NOTE
    B1="K=1.1" $CALL NOTE
    B1="K=L+M" $CALL NOTE
    B1="K=L*M" $CALL NOTE
    B1="K=L/M" $CALL NOTE
    B1="K=L" $CALL NOTE
    B1="X=L" $CALL NOTE
    B1="L=Y" $CALL NOTE
    B1="X=Y**2" $CALL NOTE
    B1="X=Y**3" $CALL NOTE
    B1="X=Y**Z" $CALL NOTE
    B1="E(J)=L" $CALL NOTE
    B1="E2(J,F)=L" $CALL NOTE
    B1="E3(J,F,G)=" $B2="L" $CALL NOTE
    B1="L=E(J)" $B2=" " $CALL NOTE
    B1="SUBROUTINE" $B2=" A=3.14 EN" $B3="D" $CALL NOTE
    B1="SUBROUTINE" $B2=" REAL A(1)" $B3=" A(1)=2.72" $B4=" END"
    CALL NOTE
    B1="SUBR. REAL" $B2=" A(500) " $B3="A(23)=7.8" $B4=" END"
    CALL NOTE
    B1="SUBR. REAL" $B2=" A(1,1) " $B3="A(1,1)=4.6" $B4="5 END"
    CALL NOTE
    B1="SUBR. REAL" $B2=" A(1,1,1)" $B3="A(1,1,1)=" $B4="1.7 END"
    CALL NOTE
    B1="SUBROUTINE" $B2=" GOTO 26" $B3=" 26 END" $B4=" " $CALL NOTE
    B1="GOTO (127," $B2="227),J " $B3=" 127 CONTI" $B4="NUE"
    CALL NOTE
    B1="X=SIN(Y)" $B2=" " $B3=" " $B4=" " $CALL NOTE
    B1="X=COS(Y)" $CALL NOTE
    B1="X=ABS(Y)" $CALL NOTE
    B1="X=EXP(Y)" $CALL NOTE
    B1="X=ALOG(Y)" $CALL NOTE
    B1="X=SQRT(Y)" $CALL NOTE
    B1="X=ATAN(Y)" $CALL NOTE
    B1="X=0  IF(Y." $B2="GT.0) X=1" $B3="IF (Y.LT.0" $B4=") X=-1"
    CALL NOTE
    B1="X=INT(Y)" $B2=" " $B3=" " $B4=" " $CALL NOTE
    B1="CALL P0" $CALL NOTE
    B1="CALL P1(X)" $CALL NOTE
    B1="CALL P2(X," $B2="Y)" $CALL NOTE
    B1="CALL P3(X," $B2="Y,Z)" $CALL NOTE
    B1="LOOP OF " $B2="DO 40 I=1," $B3="N   40 CON" $B4="TINUE"
```

```
      CALL NOTE
      S=S/42.
      WRITE(6,200) S,S
  200 FORMAT(1H //*  1.000*,2F7.1,*  MIXTURE*,/)
      STOP
      END
      SUBROUTINE P0
      COMMON/L4/X
      X=3.14
      RETURN
      END
      SUBROUTINE P1(X)
      X=1.25
      RETURN
      END
      SUBROUTINE P2(X,Y)
      Y=1.4
      RETURN
      END
      SUBROUTINE P3(X,Y,Z)
      Z=5.6
      RETURN
      END
      SUBROUTINE SAVE
      COMMON/L1/EP,TT(42)/L2/Q,P,C,N
      EP=EP+1
      TT(EP)=(Q-P-C)/N
      RETURN
      END
      SUBROUTINE NOTE
      COMMON/L1/EP,TT(42)/L3/RELFREQ,FREQ(42),FSUM,LOAD,TIJD,S,B1,B2,B3,
     CB4
      REAL LOAD
      EP=EP+1
      RELFREQ=FREQ(EP)*FSUM
      TIJD=1.E6*TT(EP)
      LOAD=RELFREQ*TIJD
      S=S+LOAD
      WRITE(6,25) RELFREQ,LOAD,TIJD,B1,B2,B3,B4
   25 FORMAT(1H ,F6.3,2F7.1,2X,4A10)
      RETURN
      END
      SUBROUTINE Q1
      A=3.14
      RETURN
      END
      SUBROUTINE Q2
      REAL A(1)
      A(1)=2.72
      RETURN
      END
```

```
      SUBROUTINE Q3
      REAL A(500)
      A(23)=7.8
      RETURN
      END
      SUBROUTINE Q4
      REAL A(1,1)
      A(1,1)=4.65
      RETURN
      END
      SUBROUTINE Q5
      REAL A(1,1,1)
      A(1,1,1)=1.7
      RETURN
      END
      SUBROUTINE Q6
      GOTO 26
   26 RETURN
      END
```

9.4.3.2. MNF timings

The MNF Fortran compiler has not been discussed extensively in this survey.

check D	¬check	
.7	.9	X=1.1
3.8	3.3	X=11
.7	1.3	X=Y
3.2	3.3	X=Y+Z
7.0	6.4	X=Y*Z
6.3	6.3	X=Y/Z
2.0	2.2	K=11
3.2	3.2	K=1.1
1.3	.8	K=L+M
6.3	6.4	K=L*M
10.7	10.9	K=L/M
.7	.9	K=L
3.2	3.1	X=L
2.6	2.7	L=Y
8.2	8.0	X=Y**2
14.5	14.3	X=Y**3
203.0	202.7	X=Y**Z
.9	.8	E(J)=L
1.3	1.3	E2(J,F)=L
.9	1.1	E3(J,F,G)=L
1.3	1.3	L=E(J)
20.2	20.5	SUBROUTINE A=1.4 END
20.6	20.4	SUBROUTINE REAL A(1) A(1)=2.72 END
20.3	20.5	SUBR. REAL A(500) A(23)=7.8 END
20.2	21.4	SUBR. REAL A(1,1) A(1,1)=4.65 END
20.5	20.6	SUBR. REAL A(1,1,1) A(1,1,1)= 1.7 END
16.4	16.9	SUBROUTINE GOTO 26 26 END
9.7	9.4	GOTO (127,227),J 127 CONTINUE
129.7	129.6	X=SIN(Y)
136.0	135.3	X=COS(Y)
3.8	3.9	X=ABS(Y)
98.2	97.8	X=EXP(Y)
118.1	117.9	X=ALOG(Y)
93.6	93.2	X=SQRT(Y)
101.8	101.1	X=ATAN(Y)
19.4	20.3	X=0 IF(Y.GT.0) X=1 IF (Y.LT.0) X=-1
5.9	5.8	X=INT(Y)
20.8	20.0	CALL P0
29.6	29.1	CALL P1(X)
32.0	31.2	CALL P2(X,Y)
34.2	34.8	CALL P3(X,Y,Z)
2.4	2.5	LOOP OF DO 40 I=1,N 40 CONTINUE
8.1	8.1	MIXTURE

9.4.4. Algol 60 timings

3pch	3½ch	4ch	4½ch	
2.9	2.6	2.2	6.2	X:= 1.1
4.8	6.1	5.3	3.5	X:= 11
1.1	2.8	5.3	2.9	X:= Y
5.4	4.4	5.3	6.4	X:= Y + Z
8.6	12.7	9.2	9.3	X:= Y * Z
11.3	8.7	11.6	11.9	X:= Y / Z
2.1	4.3	5.3	3.0	K:= 11
3.3	5.8	8.5	12.5	K:= 1.1
6.2	6.4	2.1	4.1	K:= L + M
11.2	9.7	14.6	9.9	K:= L * M
15.2	14.4	21.0	26.3	K:= L // M
4.0	21.6	5.9	2.6	K:= L
4.5	4.9	2.4	4.3	X:= L
7.8	7.4	8.4	9.5	L:= Y
11.3	10.8	11.7	8.1	X:= Y ** 2
17.9	19.4	13.1	15.6	X:= Y ** 3
342.1	346.2	252.9	260.1	X:= Y ** Z
21.9	13.7	58.5	9.7	E[J]:= L
36.1	31.0	89.1	20.4	E2[J,F]:= L
48.2	40.3	114.7	31.8	E3[J,F,G]:= L
21.5	17.9	58.4	10.8	L:= E[J]
85.6	87.6	62.7	66.6	'BEGIN' 'REAL' A ; A:=3.14 'END'
159.2	155.1	143.6	144.2	'BEGIN' 'ARRAY' A[1:1] ; A[1]:=2.72 'END'
158.6	151.3	142.5	144.1	'BEGIN' 'ARRAY' A[1:500] ; A[23]:=7.8 'END'
180.3	170.7	175.6	168.9	'BEGIN' 'ARRAY' A[1:1,1:1] ; A[1,1]:=4.65 'END'
213.4	205.9	208.4	202.8	'BEGIN' 'ARRAY' A[1:1,1:1,1:1] ; A[1,1,1]:=1.7 'END'
18.7	17.0	21.0	20.6	'BEGIN' 'GOTO' ABCD ; ABCD: 'END'
123.0	128.3	107.2	112.8	'BEGIN' 'SWITCH'SS:=PQ ;'GOTO'SS[1]; PQ: 'END'
178.3	176.6	135.5	137.1	X:= SIN(Y)
177.8	175.0	130.3	131.1	X:= COS(Y)
6.6	5.1	5.9	5.2	X:= ABS(Y)
175.8	178.5	132.7	135.3	X:= EXP(Y)
171.7	172.9	127.2	128.8	X:= LN(Y)
136.4	137.2	89.6	92.5	X:= SQRT(Y)
181.2	179.0	133.5	135.6	X:= ARCTAN(Y)
10.4	10.8	20.9	19.2	X:= SIGN(Y)
71.7	70.0	11.6	12.3	X:= ENTIER(Y)
212.7	212.1	256.3	266.8	P0
250.5	250.9	331.8	330.9	P1(X)
291.7	289.9	406.5	409.5	P2(X,Y)
331.8	328.9	489.9	483.6	P3(X,Y,Z)
39.7	39.6	25.9	26.7	LOOP OF 'FOR'I:=1'STEP'1'UNTIL'N'DO';
34.9	34.0	50.4	35.3	MIXTURE

3: Algol 60 version 3.

4: Algol 60 version 4.

pch: partial check: it is checked that final array-element addresses are within the array.

ch: each subscript is checked against its proper bounds.

¬ch: no subscript checking is done.

```
TITLE:  TIMER    (ALGOL60 EXECUTION TIME, IDEA WICHMANN)
AUTHOR:   SARA, VELDHUYZEN, 740401.
LAST MODIFIED AT 760202.
"BEGIN"
   "INTEGER"I,J,F,G,N,EP;
   "REAL" P,Q,C,FSUM,S,RELFREQ,LOAD,TIME;
   "REAL" "ARRAY" TT,FREQ[1:42];

   "PROCEDURE" SAVE;"BEGIN" EP:=EP+1; TT[EP]:=(Q-P-C)/N "END";

   "PROCEDURE" WRITE(STAT); "STRING" STAT;
     "BEGIN" EP:= EP+1;RELFREQ:=FREQ[EP]*FSUM;TIME:=_6*TT[EP];
       LOAD:=RELFREQ*TIME; S:=S+LOAD;
       OUTPUT(61,"("/,ZD.3D,2(4ZD.D),2B")",RELFREQ,LOAD,TIME,STAT)
     "END";

   "INTEGER" K,L,M;
   "REAL" X,Y,Z;
   "INTEGER""ARRAY"E[1:1],E2[1:1,1:1],E3[1:1,1:1,1:1];

   "PROCEDURE" P0; X:=3.14;
   "PROCEDURE" P1(X); "VALUE"X; "REAL"X; X:=1.25;
   "PROCEDURE" P2(X,Y); "VALUE"X,Y; "REAL"X,Y; Y:=1.4;
   "PROCEDURE" P3(X,Y,Z); "VALUE"X,Y,Z; "REAL"X,Y,Z; Z:=5.6;

   Y:=Z:=1.1; L:=M:=E[1]:=11; I:=J:=F:=G:=1; FSUM:=0;

   "FOR"X:=10000,7000,10000,26682,31210,11000,3000,500,4300,4978,
        480.9,5000,4000,500,2780,309,442,23795,15963,296,
        23795,0,59,59,39,.73,2010,94,1020,1490,
        1390,831,644,1750,591,81.9,909,788,2316,2316,
        6053,17800 "DO""BEGIN"FREQ[I]:=X;FSUM:=FSUM+X;I:=I+1"END";
   FSUM:=42/FSUM;

   OUTPUT(61,"("/,16B,"("ALGOL60 EXECUTION TIME")",2/,
                 "(" FREQ  WEIGHT   TIME  STATEMENT")",/")");
   N:=10000;

   C:=0;"FOR"EP:=1"STEP"1"UNTIL"10"DO"
   "BEGIN"
   P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO";
   Q:= CLOCK; C:= C+(Q-P)
   "END";
```

```
C:=C/10; TT[42]:=C/N; EP:= 0;

P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" X:= 1.1;
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" X:= 11;
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" X:= Y;
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" X:= Y + Z;
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" X:= Y * Z;
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" X:= Y / Z;
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" K:= 11;
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" K:= 1.1;
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" K:= L + M;
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" K:= L * M;
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" K:= L // M;
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" K:= L;
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" X:= L;
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" L:= Y;
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" X:= Y ** 2;
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" X:= Y ** 3;
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" X:= Y ** Z;
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" E[J]:= L;
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" E2[J,F]:= L;
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" E3[J,F,G]:= L;
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" L:= E[J];
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" "BEGIN" "REAL" A;
                                              A:=3.14 "END";
Q:= CLOCK; SAVE;
P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO""BEGIN""ARRAY"A[1:1];
                                                 A[1]:=2.72 "END";
```

```
    Q:= CLOCK; SAVE;
    P:=CLOCK;"FOR"I:=1"STEP"1"UNTIL"N"DO""BEGIN""ARRAY"A[1:500];
                                                    A[23]:=7.8 "END";
    Q:= CLOCK; SAVE;
    P:=CLOCK;
       "FOR"I:=1"STEP"1"UNTIL"N"DO""BEGIN""ARRAY"A[1:1,1:1];
                                                  A[1,1]:=4.65 "END";
    Q:= CLOCK; SAVE;
    P:=CLOCK;
    "FOR"I:=1"STEP"1"UNTIL"N"DO""BEGIN""ARRAY"A[1:1,1:1,1:1];
                                              A[1,1,1]:=1.7 "END";
    Q:= CLOCK; SAVE;
    P:=CLOCK;"FOR"I:=1"STEP"1"UNTIL"N"DO""BEGIN""GOTO"ABCD;ABCD: "END";
    Q:= CLOCK; SAVE;
    P:= CLOCK;
"FOR"I:=1"STEP"1"UNTIL"N"DO""BEGIN""SWITCH"SS:=PQ;"GOTO"SS[1];PQ:"END";
    Q:= CLOCK; SAVE;
    P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" X:= SIN(Y);
    Q:= CLOCK; SAVE;
    P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" X:= COS(Y);
    Q:= CLOCK; SAVE;
    P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" X:= ABS(Y);
    Q:= CLOCK; SAVE;
    P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" X:= EXP(Y);
    Q:= CLOCK; SAVE;
    P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" X:= LN(Y);
    Q:= CLOCK; SAVE;
    P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" X:= SQRT(Y);
    Q:= CLOCK; SAVE;
    P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" X:= ARCTAN(Y);
    Q:= CLOCK; SAVE;
    P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" X:= SIGN(Y);
    Q:= CLOCK; SAVE;
    P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" X:= ENTIER(Y);
    Q:= CLOCK; SAVE;
    P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" P0;
    Q:= CLOCK; SAVE;
    P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" P1(X);
    Q:= CLOCK; SAVE;
    P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" P2(X,Y);
    Q:= CLOCK; SAVE;
    P:= CLOCK; "FOR"I:=1"STEP"1"UNTIL"N"DO" P3(X,Y,Z);
    Q:= CLOCK; SAVE;

    EP:=0., S:=0;
```

```
    WRITE("("X:= 1.1")");
    WRITE("("X:= 11")");
    WRITE("("X:= Y")");
    WRITE("("X:= Y + Z")");
    WRITE("("X:= Y * Z")");
    WRITE("("X:= Y / Z")");
    WRITE("("K:= 11")");
    WRITE("("K:= 1.1")");
    WRITE("("K:= L + M")");
    WRITE("("K:= L * M")");
    WRITE("("K:= L // M")");
    WRITE("("K:= L")");
    WRITE("("X:= L")");
    WRITE("("L:= Y")");
    WRITE("("X:= Y ** 2")");
    WRITE("("X:= Y ** 3")");
    WRITE("("X:= Y ** Z")");
    WRITE("("E[J]:= L")");
    WRITE("("E2[J,F]:= L")");
    WRITE("("E3[J,F,G]:= L")");
    WRITE("("L:= E[J]")");
    WRITE("(" 'BEGIN' 'REAL' A; A:=3.14 'END'")");
    WRITE("(" 'BEGIN' 'ARRAY' A[1:1]; A[1]:=2.72 'END'")");
    WRITE("(" 'BEGIN' 'ARRAY' A[1:500]; A[23]:=7.8 'END'")");
    WRITE("(" 'BEGIN' 'ARRAY' A[1:1,1:1]; A[1,1]:=4.65 'END'")");
    WRITE("(" 'BEGIN' 'ARRAY' A[1:1,1:1,1:1]; A[1,1,1]:=1.7 'END'")");
    WRITE("(" 'BEGIN' 'GOTO' ABCD; ABCD: 'END'")");
    WRITE("(" 'BEGIN' 'SWITCH' SS:=PQ; 'GOTO' SS[1];PQ: 'END'")");
    WRITE("("X:= SIN(Y)")");
    WRITE("("X:= COS(Y)")");
    WRITE("("X:= ABS(Y)")");
    WRITE("("X:= EXP(Y)")");
    WRITE("("X:= LN(Y)")");
    WRITE("("X:= SQRT(Y)")");
    WRITE("("X:= ARCTAN(Y)")");
    WRITE("("X:= SIGN(Y)")");
    WRITE("("X:= ENTIER(Y)")");
    WRITE("("P0")");
    WRITE("("P1(X)")");
    WRITE("("P2(X,Y)")");
    WRITE("("P3(X,Y,Z)")");
    WRITE("("LOOP OF 'FOR' I:=1 'STEP' 1 'UNTIL' N 'DO';")");

S:=S/42;
OUTPUT(61,"("2/,ZD.3D,2(4ZD.D),2B,"("MIXTURE")",/")",1,S,S)
"END"
```

10. Conclusions

Two of the language implementations surveyed seem to be the most appropriate for general programming: Algol 68 and Pascal. Pascal should be considered if efficiency of input and output is crucial. Algol 68 is better if the program is logically complex and especially if it deals in complicated data structures, because of its greater internal run-time security. If sizes of arrays are to be chosen at run-time, as is necessary for many numerical applications, Algol 68 should be chosen above Pascal.

LITERATURE

[1] JENSEN, KATHLEEN & NIKLAUS WIRTH, *PASCAL User Manual and Report*, Springer Verlag 1975.

[2] *Draft Proposed ANS Fortran, BSR X3.9, X3J3/76*, in Sigplan Notices 11 (1975) 3.

[3] VAN WIJNGAARDEN, A., et al., *Revised Report on the Algorithmic Language Algol 68*, Acta informatica 5 (1975), pp. 1-236.

[4] LEARNER, A. & A.J. POWELL, *An Introduction to Algol 68 through Problems*, MacMillan 1974.

[5] NAUR, PETER et al., *Revised Report on the Algorithmic Language Algol 60*, Comm. ACM (Jan. 1963), pp. 1-17.

[6] *USA Standard Fortran (USAS X3.9-1966)*, USA Standards Institute, New York, 1966.

[7] *USA Standard Basic Fortran (USAS X3.10-1966)*, USA Standards Institute, New York, 1966.

[8] *Fortran vs. Basic Fortran*, Comm. ACM 7,10 (Oct. 1964), 591-625.

[9] HEISING, W.P., *History and Summary of FORTRAN Standardization Development for the ASA*, Comm. ACM 7,10 (Oct. 1964), 590.

[10] *Standard FORTRAN Programming Manual*, the National Computing Centre Limited, 1972, SBN 85012 063 2.

[11] KUKI, H., *Mathematical Function Subprograms for Basic System Libraries - Objectives, Constraints, and Trade-Off,* in *Mathematical Software,* edited by John R.Rice, ACM Monograph Series, Academic Press, 1971.

[12] TANENBAUM, A.S., *A tutorial on Algol 68,* Computing Surveys, September 1976.

[13] WIRTH, NIKLAUS, *On "Pascal", code generation, and the CDC 6600 computer,* Report STAN-CS-72-257, Computer Science Department, School of Humanities and Sciences, Stanford University.

[14] KNUTH, D.E., et al., *A proposal for input/output conventions in ALGOL-60,* CACM vol. 7, no. 5, May 1964, pp. 273-283.

[15] WICHMANN, B.A., *Algol 60 Compilation and Assessment,* APIC Studies in Data Processing, 10, Academic Press 1970.

[16] KOK, J. & K. DEKKER (samenstellers), *Vergelijking van Rekentijden,* Report NN 8/76, Mathematisch Centrum, Amsterdam, 1976.

[17] DE MORGAN R.M., I.D. HILL & B.A. WICHMANN, *A supplement to the Algol 60 Revised Report, the Computer Journal,* vol. 19, no. 3, 1977 August, pp. 276-288.

[18] DE MORGAN R.M., I.D. HILL & B.A. WICHMANN, et al., *Modified report on Algorithmic Language ALGOL 60, the Computer Journal,* vol. 19, no. 4, pp. 364-379.

[19] *Fortran Common Library Mathematical Routines,* CDC publication no. 60387911, revision B. (other revisions may also suffice).

[20] PAGAN, FRANK G., *A Practical Guide to Algol 68,* John Wiley & Sons, 1976, ISBN 0 471 65746 8 (cloth) or 0 471 65747 6 (paperback).

[21] VAN DER MEULEN, SIETSE G., & PETER KÜHLING, *Programmieren in ALGOL 68,* Walter de Gruyter, Berlin, New York, (band I) 1974, ISBN 3 11 004698 9, (band II) 1977, ISBN 3 11 004978 3.

[22] WIRTH, NIKLAUS, *An assessment of the Programming Language Pascal,* in *Proceedings International Conference on Reliable Software,* Sigplan Notices, v.10, n.6, June, 1975, pp.23-30.

[23] WICHMANN, B.A., *Ackermann's Function: A study in the efficiency of calling procedures*, BIT Bind 16, Hefte I, 1976, pp.103-110.

[24] WICHMANN, B.A., *Second thoughts on Ackermann's Function*, M.O.L. Bulletin Nb. 5, September 1976, pp.178-190