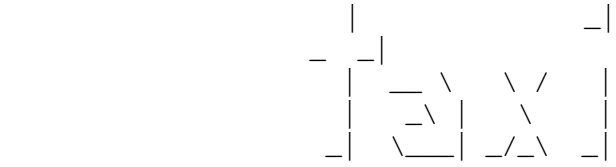
taxi MANUAL

by Antonio Maschio
programmer in Montebelluna (TV) - Italy

with the great help of Ian Jones
programmer in Bristol (UK)
and Bruce Axtens
programmer in Western Australia (AUS)

ABSTRACT



Welcome to the world of taxi! taxi is a console 'boy' ALGOL 60 interpreter, based mostly on the DEC system-10/20 ALGOL and on the ISO 1538-1984 (E) standard.

Read also the taxi man page for other more general info and the installing instructions.

Readers are encouraged to report all errors and inconsistencies found (both in the program and in the manual) to

ing dot antonio dot maschio at gmail dot com

Unlike a common taxi, this taxi won't ask you for a fare, doesn't matter how much far you can get from here. It will only ask you for a voluntary contribution, to be made through PayPal (any amount), just to support my work and let me know you like it. Donations can be directed to

tbin at libero dot it

Thanks in advance to anyone of you who will contribute. And thanks anyway to anyone of you who will download and take a trip with this taxi!

4 January 2021

This work is dedicated to the memory
of Heinz Rutishauser *(1918-1970),*
Swiss mathematician and computer scientist

1. INTRODUCTION

This manual illustrates the ALGOL language understood by taxi, which must be written always in lower letters (so no Taxi, TAXI or the like).

taxi (which means **T**onibin's **A**lgol si**X**ty **I**nterpreter) is a boy implementation of the ALGOL 60 language; ALGOL is an abbreviation of ALGOrithmic Language, and 1960 is the year when it was defined.

This document mimics the DEC system-10/20 ALGOL manual, paraphrased and integrated; it helps in using correctly the various statements, procedures and their options, using output examples for all the procedures, something that both the DEC system-10/20 ALGOL manual and the ISO 1538-1984 (E) document lacked[1].

taxi reads a textual program file and executes its instructions by interpretation, not by compilation. The interpreter is equally responsible for detecting errors in the user's source program and reporting them to the user, under the "First Error Stops Execution" Law. With modern machines, an interpreter can process ALGOL programs faster than a compiler of the Seventies and does not require the use of any backing store. Therefore, the fact taxi is an interpreter will not affect the user.

For the specific features, please refer to the various sections of this manual. For the definitions of used terms, see Section 1.2 in particular. I just want to underline here the fact that I created taxi with a specific goal in mind: security: security. I have not created it with the goal of execution speed. I rather want it to be safe. (I wouldn't want SIGSEGV errors or memory corruption messages to occur when running taxi. And if it should happen, please write to me immediately.)

NOTE: in this manual, all code examples are written in `UPPER TYPEWRITER CASE`, to differentiate the listing from the common text; anyway, consider that taxi is case insensitive for the builtin procedures and literal operators (that is `PRINT` and `print`, or `EQV` and `eqv`, are the same items), and case-sensitive for all user items, that is variables and arrays names, procedure names, string contents.

NOTE: all underlined references are references to the original documents they point to, and do not refer to parts of this manual.

1.1. A BIT OF HISTORY

ALGOL was born (let me say) on May 2, 1958, during a meeting held in Zürich between the great scientists of the American Association for Computing Machinery (ACM) and the European Society of Applied Mathematics and Mechanics (GAMM, in German words).

Those scientists were mainly F.L. Bauer, H. Bottenbruch, H. Rutishauser and K. Samelson of the GAMM, and J. Backus, C. Katz, A. Perlis and J.H. Wegstein of the ACM (Wikipedia source).

The first concept of the language was named IAL (International Algebraic Language), but it soon became ALGOL 58 when the rumours about it were spread enough. This first version is important because it is the first based on the '`BEGIN-END`' blocks model (the compound statements technique), that major languages would fully adopt years later (Pascal, C, Java, etc.) It was curious also because of the particular `FOR` cycle, with the step specified inside parentheses.

By the end of 1958 the four Universities in Zürich, München, Mainz and Darmstadt (collectively known as ZMMD) were working on an ALGOL-58 compiler for the Z22 computer. I cite this because Zürich was the

[1] I retained the term *procedure* to identify both procedures (i.e. sequences that don't return a value) and functions (i.e. sequences that return a value), as in the DEC manual, retaining the *function* term only for mathematical issues. This is of little importance, because an ALGOL programmer knows the difference that procedures are called directly, while functions cannot, and that functions can enter any expression, while procedures cannot.

place where Heinz Rutishauser worked[2].

The first report on ALGOL 60 was written on January 11-16, 1960, during a conference held in Paris. The scientists that convened were more or less the same as the 1958 meeting: L. Bauer, P. Naur, H. Rutishauser, K. Samelson, B. Vauquois, A. van Wijngaarden and M. Woodger for the European side, and J.W. Backus, J. Green, C. Katz, J. McCarthy, A.J. Perlis and J.H. Wegstein for the American side (Wikipedia source). Note the presence in the works of two important scientists, doctor J. Backus and doctor P. Naur, who will later develop the famous Backus-Naur notation for context-free grammars, still used today for the material definition of programming languages. Actually, ALGOL was maybe the first language to adopt this notation.

Anyway, the ALGOL 60 definition left several features undefined, notably the input/output procedures; this permitted the various implementers of the language some freedom in interpreting these lacking features, sometimes with too much liberty (for instance the Dartmouth Algol, which was given more or less the input/output features of the coeval BASIC language).

Many of these lacking features were discussed extensively from 1960 until 1962 when the "Revised Report on the Algorithmic Language ALGOL 60", also known as the "Revised Report", was published, as acts of the *International Federation for Information Processing* (IFIP) in 1962 in Rome (again an European town!) This revised report solves some but not all the problems of the language.

In 1972, a preliminary ISO version, based upon the "Revised Report", was published by ISO as the "Recommendation 1538 (1972)", but the IFIP technical committee refused to recognise this document as valid.

Later, in 1976, after many other meetings, the IFIP Committee ultimately approved the "Modified Report on the Algorithmic Language ALGOL 60", known since then as the "Modified Report", that was issued on The Computer Journal, Vol. 19, No. 4, Nov. 1976, pages 364÷379. The primary subject of discussion was the *recursion*, that someone reputed not suitable for ALGOL, while some others felt it was a necessary feature[3]. A paladin of the recursion was the European Computer Manufacturers' Association (ECMA), an influential institution that published many texts about programming, most of them (if not all) are available today as pdf files and freely downloadable[4].

The authoritative definition for the ALGOL 60 language is contained in the final ISO 1538-1984 (E) standard, published in October 1984, which is extensively based on the Modified Report. This Standard is nowadays abrogated, but still a milestone in ALGOL.

In my humble opinion, one of the most useful versions of ALGOL was developed by the Digital Equipment Corporation (DEC), running on the famous PDP computers. The version I consulted was the one targeted for the DEC system-10/20 (namely the AA-0196C-TK 1977 protocol). The DEC system-10/20 ALGOL was robust and sufficiently adherent to the Algol-60 philosophy, with its performant input/output procedures, that plainly coexists with the ISO protocol.

taxi uses by default the DEC protocol, with the addition of the ISO features, when not in contradiction with such protocol.

1.1.1. Restrictions with respect to the DEC System-10/20 ALGOL

taxi imposes the following restrictions with respect to DEC system-10/20 ALGOL, partly due to the fact that it runs on modern 64 bits machines, and partly due to my ineptitude:

1. Integers are limited to the range -2,147,483,648 to 2,147,483,647 (was -34,359,738,367 to 34,359,738,367). The sign appearing before the number is part of the number (optional in case it is +).

[2] More in https://en.wikipedia.org/wiki/Heinz_Rutishauser

[3] More here: https://vanemden.wordpress.com/2014/06/18/how-recursion-got-into-programming-a-comedy-of-errors-3/

[4] See here: https://www.ecma-international.org/publications/standards/Standard.htm

2. Octal numbers are limited to 11 digits (was 12)
3. ASCII constants are limited to four characters (was five), and the character enclosing the ASCII constant can be any but not the space character.
4. An array element cannot be chosen as the controlling variable in a `FOR` statement.
5. In inputting numeric data through `READ`, only digits and dots can appear in the data to be read, along with the exponent signs (`E`, `&` or `@` for real, `EE`, `&&` or `@@` for long real); spaces and tabulation characters can appear anywhere, they will be removed before recognition.
6. Logical devices do not set the end-of-file flag in input or output, acting as a circular data structure.
7. Channels 0 and 1 are not available as normal I/O channels and are always linked respectively to the standard input from the keyboard and to the standard output to the screen. This cannot be changed (as a safety measure) to have a secure channel for any useful input or output scope (i.e. as a standard error report stream). Thus, modifiable device channel numbers are limited to the range 2÷15.
8. Values bits of the procedure `IOCHAN` are not all meaningful in the sense of UNIX and in this case they are not used, but remain readable.
9. Third and fourth arguments of procedures `INPUT` and `OUTPUT` (binary or textual format type, and buffering depth) are ignored.
10. Built-in procedure names are always represented in the reserved word format; the non-reserved words enclosed in single quotes (ticks) are not allowed. Modern languages adhere to this philosophy. Moreover, the whole DEC system-10/20 ALGOL manual is written according to the reserved word format. If you have a listing in non-reserved words, you can use option `--purge` to get the correct version of the program and try if taxi can digest it.
11. This interpreter taxi is a boy interpreter. (See par. 1.1.5 "A necessary clarification".)

1.1.2. Enhancements with respect to the DEC System-10/20 ALGOL

taxi implements the following extensions to the DEC system-10/20 ALGOL:

1. The double-precision of C permits a reasonably good performance with the ALGOL `REAL` type; the `LONG REAL` type of the DEC system-10/20 ALGOL is added, equivalent to FORTRAN's double-precision, but the two types are stored in the same C double type, so the difference is only in the output picturing; thus, the user can access the power to handle double-precision real numbers even with the `REAL` type, without caring about the `REAL`/`LONG REAL` conversion issues.
2. Numeric labels are available (they are treated as any other alphanumeric label).
3. The new procedures `WRITELN`, `PRINTLN`, `CREATEFILE`, `VPRINT`, `CLOCK`, the new constants `INF`, `ERRL`, `ERRC`, `PI` and the new string procedures `HEAD`, `TAIL`, `TAKE`, `DROP`, `CONVERT` are provided.
4. Some DEC procedures have different features or scopes: `SIZE`, `CALL`, `DUMP`, `ONTRACE`, `OFF-TRACE`, `GFIELD`, `SFIELD`.
5. All the trigonometric and hyperbolic procedures and their inverse, along with `ERF`, `GAMMA` and `INTEGRAL` (to calculate a finite integral) are provided; also, new procedures for value conversion are provided: `DEGREES`, `RADIANS`, `TOLONG`, `TOREAL`.
6. All variables are instantiated with zero (if numeric) or with the null string (if strings or labels), though this is required neither by the ISO 1538-1984 (3.1.5 comma 1) nor by the DEC protocol. This is extended to the fictitious variables created to store the return value of procedures. This guarantees that no uninitialized variables are used in a program.
7. The restriction that forced variables and arrays to be declared before procedures and switches has been removed. Also, variables and arrays can be declared anywhere in the listing, even after some procedure has executed.
8. The restriction, in the definition of the arguments of a `PROCEDURE`, that forced `VALUE` items to be declared before types, has been removed; arguments and specifiers can be set in any order, before the procedure instructions. Besides, in the body of a procedure, the assignment of a referenced

variable which is passed a numeric expression is possible.

9. Third and fourth arguments of the procedure `OPENFILE` (protection code in the sense of DEC Operating System and project-programmer number) have been changed to: protection code - in UNIX sense - and content-deletion flag (i.e. if true, delete file content before opening).

10 The string procedure `CONCAT` has been enabled to concatenate all strings in the list of arguments, separated by a comma (therefore, also more than two).

1.1.3. Differences with the ISO 1538-1984

Differences with the ISO 1538-1984 Standard features are here listed:

1. Strings are enclosed in double-quotes (either using the `"` character or using two instances of the single quote `''`). The ISO 1538-1984, instead, uses consistently strings with different characters for the opening and closing. The whole examples in par. 2.6.2 were built in such a way. taxi adopts the DEC system-10/20 ALGOL point of view (modern and concrete), and strings are enclosed in quotes that must be the same for opening and closing the string. All other characters - but the double quote `"` - represent themselves, and the double-quote literal is represented with a double instance of it `""` (unless this represent the empty string). In case of strings enclosed in double instances of the single quote (`''`), a single instance of the quote character or a single instance of the double-quote character represents itself.

2 New ISO-like procedures are provided in completion to the ones described in the Environmental Block in ISO1538-Appendix 2: `INSTRING`, `INLONGREAL`, `OUTLONGREAL`, `OUTBOOLEAN`, `INCHARACTER`, `OUTCHARACTER`,

3 New ISO-like procedures for array input/output are provided, following the suggestions in the Heinz Rutishauser's book: `INARRAY` and `OUTARRAY`, for numeric arrays.

4. All procedures in the Environmental Block in ISO document in Appendix 2 are hardwired in taxi code. It's always possible to implement them in ALGOL, but their output may differ.

5. The logical operators `AND`, `OR`, `IMP` and `EQV` in the ISO 1538-1984 follow a proper syntax (cf. 3.4.5), which taxi incorporates; so `/\` is `AND`, `\/` is `OR`, `->` is `IMP` and `==` is `EQV` (the ASCII table lacks some specific logic symbols). The usage of such ISO operators is maybe not entirely portable.

6. The precedence of `NOT` is coordinated with the negative sign for any item; This means it is evaluated immediately. This has a stronger binding than what the ISO 1538-1984 requires, that is a lower precedence with respect to the arithmetic and relational operators (cf. 3.4.6.1).

1.1.4. The documented sources

Here are the links to the documents upon which taxi is based.

The last revision of the DEC system-10/20 ALGOL manual AA-0196C-TK dates back to April 1977. It can be found here:

AA-0196C-TK_ALGOL_Programmers_Guide_Apr77.pdf

The ISO 1538-1984 (E) First edition, titled "Programming languages - ALGOL 60", was published on October 15, 1984, and no other editions exist, to my knowledge. This norm is now abrogated, i.e. it has no scope and validity; firstly, because few ALGOL systems did effectively follow the ISO protocol, and secondly because ALGOL is not used anymore for serious programming. From now on, this document will be referred to as the *ISO protocol*. It can be found here:

http://www.softwarepreservation.org/projects/ALGOL/report/ISO1538.pdf

The Revised Report can be easily found on the Internet in various formats. The best version, to my knowledge and taste, edited by Peter Naur and Approved by the council of the International Federation for

Information Processing, can be found here:

http://ddhf.dk/site_dk/rc/algol/algol60.pdf

An interesting commentary on the ALGOL 60 Revised Report, by Morgan, Hill & Wichmann, can be found here:

http://archive.computerhistory.org/resources/text/algol/ACM_Algol_bulletin/1061659/p5-de_morgan.pdf

The final and definitive version of the Revised Report called Modified Report (upon which the ISO 1538-1984 (E) is modelled), which also takes into account the previous Morgan, Hill ad Wichmann commentary, can be found here:

http://hack.org/mc/texts/modified-report.pdf

If you want to read an interesting article about ALGOL 60, titled "ALGOL, a simple explanation", by Richard F. Clippinger, from the November 1962 review "Computer and Automation", this can be found here (download the entire review):

http://bitsavers.informatik.uni-stuttgart.de/pdf/computersAndAutomation/196211.pdf

The best book about ALGOL? It's the book "Description of Algol 60" by Heinz Rutishauser, written in 1967; it's freely available here (in English):

http://www.algol60.org/docs/Rutishauser_Description_of_ALGOL_60_1967.pdf

The manual of the Algol-20 environment designed at the Carnegie University in Washington DC can be found here:

https://pdfs.semanticscholar.org

The Unisys Algol manuals, printed in 2001 for a very advanced and detailed version of Algol 60, can be found here in two volumes:

Volume 1:
https://public.support.unisys.com/aseries/docs/ClearPath-MCP-17.0/PDF/86000098-515.pdf

Volume 2:
https://public.support.unisys.com/aseries/docs/ClearPath-MCP-17.0/PDF/86000734-307.pdf

There are many other manuals for the Algol 60, built by American and European (and even Chinese) Universities, and freely available on the Internet, not detailed here.

Finally, I couldn't have finished without mentioning a wonderful site dedicated to Algol, created and maintained by Henry Levkine, full of documentation, examples, tutorials, articles, algorithms and more. You can find it here:

http://www.algol60.org

I hope you will like all this.

1.1.5. A necessary clarification

The 'man-or-boy' test was designed by Donald Knuth as a means of evaluating implementations of the ALGOL 60 programming language. The scope of the test was to distinguish compilers that correctly

implemented "recursion and non-local references" from those that did not (from Wikipedia). More news here:

https://en.wikipedia.org/wiki/Man_or_boy_test

However, the stack-structure of the built-in variables storage of taxi prevents the adherence to the *man* protocol; therefore, at present, taxi is a *boy* interpreter.

I've planned to solve this, but I must observe that the current taxi model does not prevent at all the normal programming. At present, and unless a further disproof, it fails only the Knuth man-or-boy family tests.

1.1.6. Compiling taxi

To compile taxi, you can simply run '`make`' and '`make install`' in the source directory, but I suggest taking a preliminary look at the file `custom.h`, before compiling, to customize some parameters, if you need. Here they are.

1. **Default page height**
   constant `DEFPAGEHEIGHT` (default value 68)

   Use your printer page height; it's the height for simple textual files (it is a fixed size, based on the typographical characters of your printer; if you don't know this value, to retrieve it follow these steps:

   - write a textual file with 100 lines at least. This file should contain numbers (one for each line) and nothing else; void lines count, but they must be followed by filled lines; using void lines spares ink, but assure the lines count is correct and use a full line as the last forty.
   - call the file for instance `ph.txt`.
   - put recycling paper into your printer (to save new paper)
   - from the console, type
     ```
     $ lpr ph.txt
     ```
     or
     ```
     $ cat ph.txt | lpr
     ```
     If `lpr` is not installed, find out which is the line printer command; let's call it '`mylpr`' for the sake of the example, and use it as previously explained.
   - Count how many lines are printed in the first printed page, including the empty lines; this number, that should be in the range 50..70, is the number to put after `DEFPAGEHEIGHT`

2. **Default printer command**
   constant `ROUTESTRING` (default value "`lpr`")

   Leave `lpr`, if it's installed on your system, otherwise, put your own (to follow the previous example, '`mylpr`' - see point 1)

3. **Maximum number of loadable libraries**
   constant `EXTERNLIBMAX` (default value 64)

   Put here the number you need, or leave 64 (which is sufficient for almost all tasks, as far as I can think).

4. **Executable position**
   constant `CALLEX` (default value `"./taxi "`)

   If you run taxi from the source directory, leave this value; if you install taxi, write here simply `"taxi"`; remember that one space must follow the executable name; remember also that you can put here some options if you need; this is the command that invokes programs through `CALL` (see 17.5).

After changing, or not, the file `custom.h`, you can compile and install; anyway, while '`make`' requires no root privileges (provided you own the rights to work on the source directory), '`make install`' can be run only by root, or by a member of the sudoer group (with `sudo`). In case you cannot install the program, ask your administrator.

You can alternatively change the destination path in makefile, and drive the installation to a directory of your `$HOME`, to make taxi available only for you. In this case, there's no need for root privileges.

1.1.7. The Running Process

The taxi interpreter follows these steps:

1. The whole program is loaded in memory; during this phase, all comments are removed, all labels are analyzed and removed and their position is saved for later referencing, all operators constituted by more than one character are changed to a one-character symbol (for instance: `AND` becomes _); finally, all spaces out of a string are removed. The resulting compact meta-text is what taxi stores and sees.
2. The whole meta-text is parsed for determining the inner program structures (mainly `BEGIN-END` blocks, `IF-THEN-ELSE`, `FOR` and `WHILE` positional parameters, `PROCEDURE` declarations data); this causes a certain negligible runtime excess before program start but also speeds up the running phase, and this gain in speed is increased in case of cycles because the calculation for relative jumps is done once only.
3. Being an interpreter, any error occurring in the preliminary phase, any logic error (not detected by the preliminary phase) or any syntax error (for instance using = instead of `:=`) in the run-time phase stops execution by printing an error message, with the (hopefully precise) position of the error in the line that caused the error. The printed line is the full line text (not the meta-text of previous point 1). The typical run-time error scheme is as follows (supposing `CARC` is meaningless for the current program):

```
taxi runtime processor error in the program at line 13:
Illegal name.
k:=234545; I:=CARC(3.0,ss[k]);
           ^
Error code 46
```

   The indicated position may slightly differ from the correct error position. In case the error is detected in the pre-processing phase, the error scheme is slightly different, as follows:

```
taxi preprocessor error in the program at line 1:
Irregular program flow. Check blocks.
Error code 23
```

   No indicator is printed in this case, because an error in the preprocessor is a more general error, not confined in one line or position, but involving a whole structure. (For instance a `BEGIN` without `END`.)

1.1.8. Calculation issues

There are some subtle differences between a number in memory and the representation of it.

Let's take an example. If the exponents of two numbers involved in a calculation are similar and not too different, the calculation is possible, and a correct (or so) result is returned. Suppose for instance the following addition is required:

```
long real x,y;
x:=2.3&&45;
y:=4.5&&46;
outlongreal(0,x+y);
```

The difference between the two numbers is small, and the result is the correct value

```
4.73000000000000220&&46
```

(ignore rubbish at the end). But if the following is calculated:

```
long real x,y;
x:=2.3&&145;
y:=4.5&&6;
print(x+y);
```

(see the huge difference in the exponents) a poor

```
2.30000000000000071&&+145
```

is returned (again, ignore rubbish at the end). In practice, the addition is not performed at all. How come? Because `y` happens to be in the 'rumour' of `x`, and cannot be contained in the representation of it.

Another problem concerns the integer type in mixed calculations. In an integer expression, the result is accurate up to 32 bits (-2147483648÷2147483647 for signed values, i.e. 2^{31} as magnitude); if instead you put it into an expression involving real or long real values, you'll probably lose precision, halving the interval (at about -1000000000÷1000000000 i.e. 2^{30} as magnitude). Keep this in mind when performing mixed calculations involving high values numbers.

One more problem is hidden in the round-off errors in representing the number in memory. See for instance the file `preci.alg` in the `files/` directory of the installation folder of taxi: you will see that DEC and ISO represent numbers in different ways, but all suffer from some errors in the last decimals (see for instance https://en.wikipedia.org/wiki/Round-off_error).

Last, but not least, also the errors involved in repeated calculations must be taken in account, and cannot be avoided, because the round-off error accumulates, reducing the possible precision in the result. This can be said for all calculations that do repeated summing of expression (for instance the procedure `INTEGRAL`, built-in in taxi, which suffer this problem and reduce the precision to 10/11 digits). Another known problem is called *cancellation* had happens when two floating-point numbers are subtracted: this may reduce dramatically the precision of the result.

I think that an ALGOL programmer should be aware that it's practically impossible yielding *exact* result from a floating-point expression. One must hope that there are sufficient acceptable digits, or put automatically in mind that when 24.999999999 is met, this means 25.0 (the error is small, but it totally cancel the correct representation of the number).

1.2. TERMINOLOGY

Although the terms used in this manual may seem familiar to the experienced programmer, I suggest that you carefully read the following definitions, because the ALGOL jargon is not common today and can result incomprehensible or misleading.

Delimiter Word

a single, English language word that is an inherent part of the structure of the ALGOL language. Such words cannot normally be used for other purposes. Examples:
`BEGIN IF ARRAY`.

Terminator Word

a single, English language word or sign that is used to terminate a block of code. Such words are `END`, `ELSE`, semicolon.

Identifier

a name, established by a user declaration, that represents some quantity within a program or is set as a specific name for a procedure.

Label

an identifier used to mark a certain position in a program. Control of program execution can be transferred to the statement following the label. Unlike the DEC system-10/20 ALGOL, a numeric label, similar to a FORTRAN statement number, is available in taxi ALGOL.

Procedure

part of a program, which may be invoked by *calling*. In general, an indefinite number of parameter can be supplied as arguments. No result is be returned. A procedure may be invoked as a command in the listing.

Parameter

See also Procedure.
A Formal Parameter is an identifier, used within the procedure, that represents the argument supplied when the procedure is called.

2. PROGRAM STRUCTURE

2.1. BASIC SYMBOLS

An ALGOL program consists of a sequence of symbols from the ASCII character set. The meaning of individual characters given in Table 2-1 is much the same as in other high-level languages.

Table 2-1
Basic Symbols

Symbol	Meaning or Use
A–Z a–z	Used to construct identifiers for variables and procedures
_	Underscore. Used to construct identifiers and delimiter words. Also used as AND token
0–9	Decimal digits; used to construct numeric constants and identifiers.
+ –	Arithmetic addition/subtraction operators.
*	Arithmetic multiplication operator.
/	Arithmetic division operator.
\	Arithmetic integer division operator.
÷	Arithmetic integer division operator (ISO)
^	Arithmetic exponentiation operator.
( )	Parentheses; used in arithmetic expressions and to enclose parameters in procedure specifications and calls.
[ ]	Square brackets; used to enclose subscript bounds in array declarations, and array subscript lists.
,	Comma; general separator, placed subscripts, procedure parameters, lists, between array items, in switch lists.
.	Decimal point; used in numeric constants and byte subscription Also, used as a readability symbol in identifiers.
;	Semicolon; used to terminate statements.
:	Colon; used to indicate labels, and separate lower and upper bounds in array declarations.
= #	Equality and non-equality for arithmetic/string comparisons.
< >	Less than, greater than.
& @ E	Introduces exponent in floating-point numbers.
’	Opening and closing string quotes when used doubled ’’.
"	Opening and closing string double-quotes.
!	Comment (not portable;) also used as IMP token.
% $	Octal and ASCII constants.
~	Tilde; Logical NOT (ISO)
`	Inverse tick; REM token

NOTE: the `E` symbol for the exponent in floating-point numbers was not a standard for ALGOL. The DEC system-10/20 ALGOL could understand numbers read from a file written with `E` (if coming for instance from a FORTRAN output) or `D` (if coming from a FORTRAN output with double-precision numbers), but its primary input/output characters were only `&` or `@`. taxi is consistent in all the environments, so a number read from a file or keyboard may have `E`, `&` or `@` or `D` as the exponent symbol, and you can choose the exponent to be printed between `E`, `&` or `@` using the option `--exp=X` from the console[5], where `X` is the chosen ASCII character. The default is `&`.

2.2. COMPOUND SYMBOLS

Compound symbols consist of two adjacent basic symbols. Any intervening spaces or tabs do not affect their use. The compound symbols are shown in Table 2-2.

Table 2-2
Compound Symbols

Compound Symbol	Usage
:=	Assignment (also <-)
**	Arithmetic exponentiation operator (alternate form)
<=	Less than or equal to (also =<)
<>	Not equal (also ><)
>=	Greater than or equal to (also =>)
/\	Bitwise AND (ISO)
\/	Bitwise OR (ISO)
->	Logical IMP (ISO equivalent) (also >>)
==	Logical EQV (ISO equivalent) (also <->)
&& @@	Used for the LONG REAL output format
~=	Not equal (also !=)

NOTE: the ← character as an alternative to the assignment symbol is implemented as <−, because modern keyboards don't have this key anymore. (The symbol nowadays is rather associated with the backward key function.)

2.3. RESERVED WORDS

Certain letter combinations are reserved as part of the structure of the language and cannot be used as identifiers or part of identifiers.

For instance, to determine the `BEGIN−END` levels, an important phase in the pre-processor, taxi looks for each `BEGIN−END` pair, storing the positions and other data, useful to the run-time phase to know where to jump. But a variable or a procedure named or containing `BEGIN` or `END` will influence the pre-processor, which will certainly fail in finding all pairs coupled.

The same can be said for instance for `WHILE`, `FOR`, `DO` and so forth.

Table 2-3 contains a list of all the reserved words used in taxi.

[5] I omit the output with `D` because it's not Algol and definitely out of date.

Table 2-3
Reserved Words

; (semicolon)	INCLUDE
ARRAY	INTEGER
BEGIN	LABEL
BOOLEAN	LONG REAL
COMMENT	OWN
DO	PROCEDURE
DUMP	REAL
ELSE	STEP
END	STRING
EXTERNAL	SWITCH
FOR	THEN
FORWARD	UNTIL
GO TO	VALUE
IF	WHILE

The semicolon is not an identifier of course; it was included in this list because it is substantial, and cannot be employed in purposes other than as a terminator.

The `LONG REAL` terms are reserved only if used consecutively, as reported. The word `LONG` can be used as an identifier until it is never used before `REAL`.

A special class of reserved words are the terminator delimiter words (a subset of the previous list, actually), that greatly influence the program structure. In Table 2-4 they are fully explained.

Table 2-4
Terminator Words

Terminator Word	Usage
END	Used to terminate a block begun with BEGIN; it may be followed by any free form text, and terminated by a semicolon or dot; the outermost block, called the primary (or root) block, must be terminated by a dot; all the innermost by a semicolon.
ELSE	Used to terminate the THEN part of an IF-THEN-ELSE clause.
;	Semicolon; used to terminate any statement; if the statement is followed by END or ELSE, the semicolon is not mandatory.

All the words in Tables 2-3 and 2-4 should be used neither for identifiers nor for part of identifiers, or the structure of the program could be greatly ruined.

All other identifiers (mainly for procedure names) can be used to build identifiers. (For instance you can redefine your `PRINT` routine, that at all effects will substitute the default one, or you can build your `SIN()` procedure.)

2.4. USE OF SPACING AND COMMENTARY

The readability of ALGOL programs can be enhanced by the judicious use of spacing, tab formatting, and commentary. Spaces and tabs may be used freely in a source program without constraints. All spaces in the

source (except for those inside a literal string) are removed before interpretation, thus you can insert spaces or tabs wherever you want.

Comments are introduced by either the word `COMMENT` or the symbol `!` (available also in the DEC system-10/20 ALGOL, but not necessarily in other implementations of ALGOL). Such a comment may appear anywhere in a program. All such comments extend to all the lines they are spread to and must be terminated by a semicolon, to inform taxi that the interpretation must continue.

Comments in `taxi` are removed before execution because their scope is limited to the source code (but retaining the original lines numbers, for error recognition). So you can write anything in the comments, also using extra ASCII characters or special keyword names, dots, colons and punctuation symbols (all but the semicolon). Refer to Sections 1.1.7 and 19 for the description of the processing phases.

If the first character in a line is a dash, the line is not loaded. This permits to avoid the whole line while debugging a program. The gvim syntax colouring file shows these lines as pure comments. This feature is a proper taxi feature, not belonging to the DEC system-10/20 ALGOL nor the ISO document. If you need to comment a line, ensure the line is not a continued-line (see the following Section).

There is a last useful comment: after the dot appearing after the last `END`, any text (semicolon included) is ignored, so that you can put here results, comments, algorithms, descriptions, observations, in textual form and free format.

2.5. PROGRAM LINES STRUCTURE

In general, in considering one single line of a program, taxi ignores interspersed blanks, tabs and indenting (as it should be in ALGOL); besides, as usual in ALGOL, the various cycle procedures can be *broken* in more lines, without affecting their work. Nonetheless, some rules must be observed:

* If a program line ends with a comma, it is automatically linked to the following line. This feature, called *continued-line mode*, is a classic feature in ALGOL, which is famous for its mathematical procedures with an endless list of arguments, that couldn't' fit a single line. The composed line inherits the line number of the starting line (this number is the one printed in the error message), and all the subsequent lines are emptied.

* Mathematical or string expressions should stand on the same line, and broken only on the first comma available (the continued-line mode exposed in the first point). This means that a phrase like

```
WHILE I>0 DO
```

  can be broken, as in:

```
WHILE
I>0
DO
```

  but the mathematical conditional expression cannot be broken, otherwise the algebraic parser will fail to evaluate the terminal condition. This is not too stringent a rule, because who'd ever want to break such a condition?

* The conditional assignment rule (see Section 14), even if it *looks* like an `IF-THEN-ELSE` structure, is nonetheless a mathematical/string/label function, that drives the result basing on a specific two-way condition. Therefore, it follows the previous rule and cannot be broken, unless on a possible continued-line break.

* `SWITCH`, its array name and the labels list must reside on the same line; to break the list, the continued-line mode can be used.

* Procedures cannot be detached of their arguments list, which may be long; to break such list, the continued-line mode can be used. The opening parenthesis must lie on the same line of the procedure name and separated only by blanks, in order to be distinguished from the associated fictitious variables list.

3. IDENTIFIERS AND DECLARATIONS

3.1. IDENTIFIERS

An identifier must begin with a letter, followed by a series of letters or decimal digits or dots. An identifier cannot contain more than 64 characters.

RULES:

1. There is no implied type attached to an identifier (that is, suffixes are not meaningful, like `$` for strings in BASIC).

2. All identifiers in a program (included label variables) have to be "declared" before usage.

3. Identifiers cannot be set equal to or contain any of the reserved words in Table 2-3 and 2-4; besides, variables names should neither start or be set equal to the name of an argumentless procedure. Ignoring these rules can severely interfere with the text interpretation in the pre-processing phase or in the run-time phase.

4. The case of letters, in an identifier name, matters, that is `AVOC` and `avoc` are two distinct items (variable or procedure).

For example, the following are valid and distinct identifiers:

```
I
i
Alpha
BAND
J4k5
HOUSEHOLDERTRIDIAGONALIZATION
```

Note that `I` and `i` are distinct, and that `BAND` contains the relative reserved word `AND`, but it's a valid identifier name nonetheless. The following are not valid identifiers:

`4P`	it does not begin with a letter
`1`	it is a number
`AND`	it is equal to a literal operator
`BEGIN`	it is equal to a reserved word
`TREND`	it contains a reserved word

As already said, spaces are removed from the source, and this can be used to enhance the creation of the identifier; the following are valid identifiers:

```
VELVET UNDERGROUND,  ONCE AGAIN,  TWO EPSILON
```

which will be seen and used with the names

```
VELVETUNDERGROUND,  ONCEAGAIN,  TWOEPSILON
```

but can be referred to in any place with the spaced declared identifiers.

taxi (as the DEC system-10/20 ALGOL) enables the use of a decimal point as a "readability symbol" in the names of identifiers (even for label names). These readability symbols can appear anywhere but before the

first letter, and are ignored by the interpreter. Thus:

```
ONCE.AGAIN,  PI.BY.TWO,  ALPHA3.5,  BETA.22,  X.
```

have the same effect respectively as

```
ONCEAGAIN,  PIBYTWO,  ALPHA35,  BETA22,  X
```

The difference with spaces is that the readability symbols remain in the meta-text, i.e. they are not removed from the program, useful when using the debug feature.

NOTE: taxi removed the restriction of the DEC system-10/20 ALGOL that the decimal point must appear only between two adjacent alphabetic characters.

NOTE: the readability symbols may also appear in procedure names: `PRINT.OCTAL` has the same effect as `PRINTOCTAL`.

3.2. VARIABLES DECLARATION

A declaration reserves an identifier to represent a particular quantity used in a program. Such declarations are mandatory in ALGOL.

At any particular point during program execution, the form of the variable or quantity associated with the identifier depends on the type of variable. The type of variable is determined by the type of identifier which represents it.

There are six types of scalar variables, that is, variables which contain a single value:

Integer
Real
Long Real
Boolean
String
Label

Integer, real, and long real variables are capable of holding numerical values of the appropriate type (and only of that type). The range of values is as follows:

- integer type: -2147483648 to 2147483647
  (see also the paragraph "Calculation Issues")

- real and long real type: approximately -1.7&308 through 1.7&308;
  values less than approximately 1&-16 are represented by zero but their value is retained, and when not lesser than 2.3&-308 in magnitude, they are recognized and correctly interpreted in math expressions. For instance, the simple piece of code:

  ```
  REAL A,B; A:=1.3&-180; B:=1.3&180;
  PRINT(A*B);
  ```

  returns correctly the result 1.69. (see also the paragraph "Calculation Issues")

- Boolean type: true and false; Boolean variables (similar to FORTRAN's Logical variables) can hold a Boolean quantity, which is usually one between the states TRUE or FALSE but, in general, can be any integer value (where zero is FALSE, anything else is TRUE). In taxi a Boolean type is a numeric

type. When printed, they return the token 'true' or 'false'.[6]

Should you need a numeric output (0 for false and, say, 1 for true), you should not use the Boolean type but rather the integer type; the interaction with any other Boolean variable (that is, 0 x Boolean = 0, -1 x Boolean = Boolean) is maintained. The only remarkable difference is that when you print an integer variable, used as Boolean, 0 or -1 is printed and not 'false' or 'true'

- string type: string variables are somewhat more complicated. The user is referred to Chapter 13 for a full description of the subject.

- label type: label variables hold the reference name of a label, and are used as the target of GOTO procedures.

All of the types of the above variables must be declared for use by preceding a list of the identifiers to be used by the appropriate delimiter word for their type. Throughout this manual, a "list of items" consists of those items arranged sequentially and separated by commas.

Examples:

```
INTEGER I,J,K;
REAL X,Y;
LONG REAL DOUBLE,P,Q,ELEPHANT;
BOOLEAN ISITREALLYTRUE;
STRING S,T;
LABEL LAB;
```

Notes:

1 In taxi, variables can be declared anywhere, even at the last line of the program. A strict observation of the ALGOL rules should impose that any variable and procedure must be declared before any other statement in the block. So, the fact you can override this behaviour is an enhancement, but you can ignore it, if you want to obey to the general rule.

2 In taxi, if you name a variable with a name already in use, in the same memory space, a warning appears:

```
Warning: Identifier <id> at line <n> already declared in line <n>.
```

where <id> is the name being declared and <n> is the line where it was declared. This warning does not prevent the replication of the name, but this is done at your risk because, if you declare a variable i and an array i, though they are effectively two different items, they can be confused by having the same name, the latter hiding the former. Of course, if variable i is declared in the main body, and the variable i is declared into a procedure, they cannot be confused. It's actually true that the variable i in the procedure *hides* the variable i in the main body, but this is an user choice (even gcc does not warn about this kind of declaration).

3.3. A NECESSARY WARNING ABOUT VARIABLES NAMES

A variable identifier with the name equal to the name of a procedure with no arguments can confuse taxi.

For example, it's possible redefining NEWLINE (an existing argumentless procedure); at any time, by

[6] Boolean variables are stored internally as 0 for false and -1 for true, and they bear those values when inserted in any mathematical expression, which can nonetheless involve any other mathematical type.

calling NEWLINE, the new procedure will be used and not the default one. So far, so good.

But if you name NEWLINE a variable (for example a Boolean), calling:

```
BOOLEAN NEWLINE;
NEWLINE:= FALSE;
PRINT(NEWLINE);
```

the Boolean variable named NEWLINE will be correctly created and assigned (because the assignment recognizes variables only); however, in the printing phase, taxi will see the default procedure NEWLINE as an unsuitable argument for PRINT, and will complain, printing an error message.

Besides, operator names with alphabetic characters deserve special clarification: literal operator names, like AND, are not reserved words, so you might think they can be used as normal identifiers, and you can really do it, taxi will not prevent such use.

This is anyway like a two-edged sword; if you use the following sequence:

```
REAL AND;
AND:=3.4;
PRINTLN(AND);
```

you will correctly get 3.4 printed on the screen. But if you should add spaces, like in the following example, perhaps to make it clearer:

```
PRINTLN( AND );
```

you will get:

```
taxi runtime processor error in the program at line 4:
Illegal formula.
PRINTLN( AND );
     ^
Error code 31
```

because the interpreter was asked to print an operator, and not any math or Boolean expression. Operator names with alphabetic characters are recognized as such only if enclosed by blanks (space or tabulation).

The consequence of this is that you should avoid using the operators' names as identifiers themselves; anyway, you can use them as *part* of an identifier. (For instance, you can use BAND, since the operator AND is not recognized as an operator, because it is not entirely enclosed by blanks.)

4. CONSTANTS

4.1. NUMERIC CONSTANTS

There are three forms of numeric constants[7]:

1. Integer constants

2. Real constants

3. Long Real constants

Note: with regard to the symbol used to divide the integer part from the fractional part of a number, the dot (ASCII 46) is the universally used and accepted symbol (e.g. `III.FFFFF`), and so it is here. But taxi also accepts and parses every instance of the · symbol outside of strings used to signal multiplication (it's the central dot, ASCII 62 and 73, called also *dotmath*), substituting it with the common asterisk. If you happen to copy and paste a listing from a book of the Sixties to your file, you're likely to see this central dot, used in those times to symbolize typographically the multiplication. It's a painful task parsing manually the program to see where this is used in place of the asterisk (I speak by experience), so I enabled a silent substitution, harmless and useful.

4.1.1. Integer Constants

Integer constants consist of a series of adjacent decimal digits, subject to the constraint that the number represented must be in the range of integer numbers.

NOTE: any preceding sign, which is not the minus sign, that appears in the program is considered part of the constant. A plus sign is read but ignored.

Examples:

```
3

-24

+9276541
```

See also Section 4.4 about the ASCII constants, and Section 1.1.8 related to problems in practical calculations.

4.1.2. Real Constants

Real constants consist of a decimal number (containing either an integer part, or a fractional part, or both) followed by an optional exponent. The exponent consists of either the `&` or `@` symbol followed by an optionally signed integer (`E` or `D` are also accepted in reading). This has the effect of multiplying the decimal number by the power of ten specified in the exponent. If no decimal number appears, a value of unity is assumed. If the value of a decimal number is 1, then this may be omitted, and the real constant solely represented by the exponent (which, in this case, can be `&` or `@` only) - see the last example in this section.

Examples:

```
Literal             Stored              Printed
```

[7] The terminology used in this manual reflects the original one, but I must observe here that *constants* which are referred to in this chapter are actually to be considered (in modern language) 'literal numbers'. There is no `CONSTANT` specifier in ALGOL for creating instances of unmodifiable entities like C constants, for instance.

```
3.141592653589793   3.141592653589793   3.141592654&+0
.0001               0.0001              1.000000000&-4
43.7E4              437000.0            4.370000000&+5
5@-3                0.005               5.000000000&-3
&-6                 0.000001            1.000000000&-6
```

NOTE: a number can be entered as a pure exponential part only in case of `&` and `@`: for instance, `&3` is equivalent to `1&3` and `@-1` is equivalent to `1@-1`; instead, `E1` or `D3` could be confused as variables identifiers, and so they are not considered literal numbers.

4.1.3. Long Real Constants

Long real constants are used to represent numeric quantities to approximately twice the precision available with real numbers: about seventeen decimal digits. Long real constants are formed by writing a real constant in floating-point form, by replacing the `&` or `@` by `&&` or `@@`. `E` and `D` are accepted in reading.

Examples:

```
Literal       3.14159265358979323846&&0
Stored        3.141592653589793
Printed       3.1415926535897931&&+0

Literal       12@@-3
Stored        0.012
Printed       1.20000000000000107&&-2
```

Garbage numbers (dust) may appear in the final part of a long real, due to the C-double format. See last chapter.

4.1.4. Actual numeric storage

All numbers are stored in a C 'double' memory space. That is number of type Boolean, integer, real and long real are all converted to the same C 'double' format when stored.

This offers great flexibility in treating numbers because there is only one type to check, evaluate, increment, decrement; this avoids completely the loss of precision in arithmetic expressions involving long real variables and real constants, thus making easier the conversion of programs to use long real variables. Moreover, the C 'double' type can store completely the integer quantity up to the maximum signed value, so that calculation involving only integers and Booleans, is precise to the last digit.

The disadvantage is the loss of precision if integer numbers are involved in calculations in which real and long real numbers are also involved, even if the precision is at least 2^{30}, according to my studies, which is sufficient for most cases.

4.2. OCTAL CONSTANTS

Octal constants consist of the symbol `%` followed by a series of octal digits. Up to 11 significant digits may appear (leading zeros are ignored); these digits are right justified.

Examples:

```
%7777777774

%0470
```

Octal constants can be used in any expression where an equivalent decimal number is required.

4.3. BOOLEAN CONSTANTS

Boolean constants consist of numeric values whose inequality of zero is considered. If they are zero, they are printed as `false`, otherwise as `true` (equivalent to the values 0 and -1 respectively).

Boolean numbers can enter any calculation because they are stored as numbers, not as labels.

taxi offers the two constants `TRUE` and `FALSE` that can substitute respectively -1 and 0, and simulate optimally the Boolean constants.

4.4. ASCII CONSTANTS

Up to four ASCII symbols can be packed right-justified to give an integer-type constant. The format is a dollar sign (`$`), followed by up to four ASCII symbols enclosed within a delimiting symbol pair.

The leading delimiter symbol immediately follows the `$`, and can be any readable character, but not an invisible one such as the space character.

Examples:

```
Text        Decimal Value            Octal Value
-------------------------------------------------
$_A_                    65             %0000000101

$/1234/          825373492             %6114431464

$|ANNA|         1095650881             %10123447101

$M~~~~M         2122219134             %17637477176
```

The greatest integer generated by this algorithm is 2139062143, returned by four figures of ASCII 127. ASCII characters in the range 128-255 cannot be interpreted correctly by taxi. The lower integer generated by this algorithm is 1, returned by one figure of ASCII 1 (the `CSOH` character).

NOTE: the number-generation algorithm differs from the one used by the DEC system-10/20 ALGOL.

NOTE: the space character cannot be used nor as an ASCII constant delimiter neither as an ASCII figure, because spaces are removed before execution (see section 2.4).

4.5. STRING AND LABEL CONSTANTS

String constants allow the user to store any reasonable length string of ASCII characters within a program. String constants are typically used to output a message during the execution of the program or as values assigned to string variables.

The string of symbols is enclosed within quotes, the double-quote character **"** or two instances of the single-quote character **''**. There are restrictions on the symbols that may appear within the string:

1. `[` and **"** cannot appear alone inside a string.

2. `[` must appear only if properly paired with `]`.

3. Double occurrences of `[[ ]]` **;;** and **""** respectively are represented as `[ ]` **;** and **"**

4. Each opening " must be closed by an occurrence of ".

5. Each opening '' must be closed by an occurrence of ''.

Single square brackets are used to enclose symbols that have specific effects when the string is output. These are discussed in Paragraph 16.9.2.

Examples:

```
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
"REMEMBER THAT SPACES ETC. ARE SIGNIFICANT"
"[P5C]INPUT DATA: [5C]"
''""A[[I]] := 0.1;;""''
```

The latter example builds the following string: "A[I] := 0.1;", and has the double quotes as constituent parts of the string itself.

As stated in the ISO 1538-1984, (2.6.2) two strings in two consecutive lines are joined together implicitly. The ISO 1538-1984 also states that two strings should join even if in the same line, but the double-double-quote mechanism for inserting the double-quote character into the string (a heritage of the DEC system-10/20 ALGOL) forbids this. A minor issue.

Labels are a particular version of strings. They can contain only letters, numbers or dots, and they can be assigned as common strings wherever in the listing:

```
LABEL LAB;
LAB:="ERR";
```

This assignment lets use GOTO LAB or GOTO ERR with the same effect, but the destination can be changed if LAB is changed.

When passing literal label arguments to a procedure, the label may be naked; the system can detect if a naked name is a label variable (containing some value) or a naked literal label name. E.g. the following procedure calls are identical:

```
PRINT(SQUAREROOT(X,"ERR"));
PRINT(SQUAREROOT(X,''ERR''));
PRINT(SQUAREROOT(X,ERR));

LABEL LAB;
LAB:="ERR";
PRINT(SQUAREROOT(X,LAB));
```

See chapter 7 for the specific usage of labels.

5. EXPRESSIONS

5.1. ARITHMETIC EXPRESSIONS

Arithmetic expressions are written in a form similar to that used in FORTRAN, BASIC, C and many other high-level scientific computer languages. The usual algebraic rules concerning the precedence of operators and parentheses are applied. (See Table 5-4 for the complete precedence list.)

The basic arithmetic operators are:

+	addition
−	subtraction
/	ordinary division
\	integer division (also ÷)
*	multiplication
^	exponentiation (also **)

There are two additional operators, DIV and REM (the latter being also available as MOD), that indicate integer division and remainder, respectively, and these have the same precedence as ordinary division. Within the precedence scheme, the order of evaluation is always from left to right. For example:

```
X - Y - Z means (X - Y) - Z
```

and

```
I DIV J REM K means (I DIV J) REM K
```

Unlike FORTRAN, when an ordinary division of one integer constant by another is performed, the real result is not rounded to an integer value.

The difference between the various types of division is clarified by the following examples:

`7/4` yields a result of 1.75, whereas

`7 DIV 4` yields a result of 1 (the same for 7 \ 4), and

`7 REM 4` yields a result of 3

The interpretation of integer division for negative integers follows:

```
Let M, N>O, then

-M DIV N = M DIV (-N) = -(M DIV N)
-M DIV (-N) = M DIV N
```

The integer remainder operator, REM, is defined so that for all integers M, N:

```
M REM N = M - N * (M DIV N)
```

NOTE: the usage of ÷ (ISO flavour) instead of `DIV` (DEC flavour), to perform the integer division operation is also available.

5.1.1. Identifiers And Constants

Arithmetic expressions consist of operands, that is, identifiers and constants, of the three types, integer, real and long real plus one fourth (Boolean), together with arithmetic operators `+ − * / \ ÷ DIV REM MOD` and `^`, with parentheses where necessary.

Since automatic conversion takes place as necessary when an expression is evaluated, the user may freely mix the three different types of identifiers and constants.

5.1.2. Special Procedures

Three special procedures are provided for use in arithmetic expressions. The first is the transfer procedure `ENTIER`, which converts a number of any type into an integer quantity defined as the largest integer value not exceeding the argument.

Thus

```
ENTIER(3.5) = 3
```

and

```
ENTIER(−3.5) = −4
```

The special procedure `ABS` yields the absolute value (also known as the *modulus*) of its argument, which can be a number of any type.

Thus

```
ABS(−3.5) = 3.5
```

and

```
ABS(−3) = 3
```

The special procedure `SIGN` is the signum function whose argument can be a number of any type. The result is always of integer type, being -1, 0 or +1, depending on whether the argument is negative, zero, or greater than zero, respectively.

Thus

```
SIGN(−3.5) = −1
SIGN(O) = 0
SIGN (3.5) 1
```

Examples of simple arithmetic expressions follow:

```
X
I+3
X*Y/Z
P+Q/R
X2 + Y
XJ − 4
J + ENTIER(K−2)
```

```
SIGN(ENTIER(J/K) + 1)
(X + Y) * (-I)
```

5.2. BOOLEAN EXPRESSIONS

Boolean expressions involve Boolean identifiers, Boolean and octal constants, arithmetic conditions, and Boolean operators interspersed in an order similar to that of arithmetic expressions.

5.2.1. Boolean Operators

There are five Boolean operators listed here in decreasing order of precedence:

`+`	addition
`NOT`	(unary operator, also `~`)
`AND`	(also `/\`, bitwise)
`OR`	(also `\/`, bitwise)
`IMP`	(logical implication, also `->`)
`EQV`	(logical equivalence, also ==)

The operator `NOT` is a unary operator that complements a Boolean quantity in the same way that a unary minus sign negates an arithmetic quantity in an arithmetic expression, so that `FALSE` is changed to `TRUE`, or vice versa. `NOT` has the same precedence of the minus sign for numbers. This means that an expression such as

```
NOT I AND 0
```

is not interpreted as `NOT(I AND 0)` but as `(NOT I) AND 0`; it's the very same case of the minus sign, where

```
-1 = 0
```

is never interpreter as `-(1 = 0)` and the minus has the usual same position learned at school.

Table 5-1 gives the result of `A OP B` where `OP` stands for one of the Boolean operators `AND`, `OR`, `IMP`, or `EQV`, for all values of `A` and `B`.

Table 5-1
Boolean Operators

A	B	A AND B	A OR B	A IMP B	A EQV B	NOT A
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE
TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	
FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	

Besides, the following theorems hold:

`A EQV B` is equivalent to `(A IMP B) AND (B IMP A)`

`A IMP B` is equivalent to `NOT A OR B`

5.2.2. Evaluation Of Boolean Variables

Boolean variables have a value consisting of any pattern of bits, rather than be confined to the values `TRUE` and `FALSE`. The logical operations operate on a bit-by-bit basis according to the preceding rules.

The actual test employed to determine the truth of a Boolean expression such as

```
B AND C
```

is to evaluate it and regard it as true if the value is nonzero, that is, at least one bit of the associated integer value is set, otherwise, regard as false.

This is particularly important when octal constants are used in Boolean expressions. For example, if the user wishes to test a particular bit in a Boolean variable, a suitable decimal or octal constant can be used:

```
B AND %1
```

is a Boolean expression that is true if and only if the bottom (least significant) bit of `B` is a one.

5.3. ARITHMETIC CONDITION TESTS

Arithmetic conditions are used as operands in Boolean expressions. They consist of two arithmetic expressions coupled with a comparator. The comparator, which decides the particular type of test to be performed on the two expressions, is one of the following:

<	less than
<=	less than or equal to
=	equals
>	greater than
>=	greater than or equal to
#	not equal to

The operator # can also be written as the infix forms `NOTEQUAL`, <> and ~=.

Such an arithmetic condition can be regarded as true or false according to whether the condition specified by the comparator is met when the arithmetic expressions on each side are evaluated. The resulting condition is a Boolean expression.

The following examples of Boolean expressions, shown in Table 5-2, also involve arithmetic conditions.

Table 5-2
Boolean Expressions

Expression	Meaning
NOT B	NOT B
NOT A=B	(NOT A)=B
B AND NOT C	B AND (NOT C)
A OR B AND C	A OR (B AND C)
B EQV X<Y	B EQV (X<Y)
X+Y<Z AND B OR P=Q	(((X+Y)<Z) AND B) OR (P=Q)

5.4. FOREIGN OPERATORS

Since ALGOL birth, all the different compilers/interpreters conceived their syntax for math and logical operators even if, in the Modified Report, a sort of standardization were applied.

On this premise, taxi enables an extra conversion procedure, in the pre-processing phase, to substitute the foreign operators that don't agree with DEC or ISO forms, with the default ones.

If your ALGOL listing comes from an external source, before working out figuring what operator is to be changed, this feature may help in executing the code without having to change the code. If you should know about other operators not in the previous list, write me: I will add them to the reduction list.

In the following Table 5-5, the list of the operators that are converted and the taxi equivalent is represented:

Table 5-3
Conversions

Symbol	Meaning or use
EQL	= (equal operator)
!=	# (not equal operator)
><	# (not equal operator)
NEQ	# (not equal operator)
% (*)	REM (remainder/modulus operator)
>>	IMP (logic implication operator)
IMPL	IMP (logic implication operator)
<->	EQV (logic equivalence operator)
EQIV	EQV (logic equivalence operator)
LSS	< (less than operator)
=<	<= (less than or equal operator)
LEQ	<= (less than or equal operator)
=>	>= (greater than or equal operator)
GEQ	>= (greater than or equal operator)
GTR	> (greater than operator)

(*) if not immediately followed by a digit.

5.5. MACHINE TOKENS

The run-time interpretation of the source program is preceded by a preparatory phase in which the text is parsed and analyzed. During this phase, all operators constituted by more than a single character are converted to a single character, for the algebraic parser's sake. Thus, what the algebraic parser sees is not the textual or compound symbols operator, but a specific ASCII code token. Since these tokens are common

ASCII characters, the usage of these in place of the original operator form is not forbidden, but it's strongly discouraged, because these could interfere with the parser phase (and are surely not portable).

The following Table 5-3 explains this in full. Please note that the ISO character ¬ is not available as a basic ASCII character, as well as ≡.

Table 5-3
Machine tokens

Operator (DEC)	Operator (ISO)	Machine token
#	≠	#
:=	:=	©
DIV	÷	\
REM		` (inverse tick)
MOD		` (inverse tick)
<=	<=	{
>=	>=	}
NOT	¬	~ (tilde)
AND	/\	_ (underscore)
OR	\/	| (vertical bar)
IMP	->	!
EQV	≡	?

These Machine tokens are the ones you see in debugging. (See chapter 19.)

5.6. INTEGER AND BOOLEAN CONVERSIONS

An integer quantity can be converted to a Boolean quantity through the dummy procedure `BOOL`. Similarly, the dummy procedure `INT` converts a Boolean quantity to an integer quantity. If only integer values are used as arguments of these procedures, they remain unchanged.

These procedures were conceived mostly for semantic correctness. Thus:

```
BOOL(I)
```

can be regarded as a Boolean operand, and

```
INT(B)
INT(%400000000000)
```

as an integer operand.

If a real or long real number is used as an argument, the decimal part is truncated, and the integer part is used as specified above.

6. STATEMENTS AND ASSIGNMENTS

6.1. STATEMENTS

The statement is the basic operational unit in ALGOL 60 and describes an operation, such as an assignment, to be performed at run time, or a single procedure.

6.2. ASSIGNMENTS

Assignments convey the value produced by the execution of an expression to a destination variable of the appropriate type. This is done by writing the destination identifier, followed first by the symbols **:** and = and then by the expression to be evaluated. Thus

```
X := Y + Z
```

causes the result of the addition of the values contained in the variables `Y` and `Z` to be placed in the variable `X`.

When an assignment is made to a variable type differing from that of the result of the expression, a type conversion is performed only when the assignment is made to a Boolean or an integer variable, by rounding (see ahead).

Boolean, integer, real and long real expressions can be assigned to variables of any of these four types, but not to any other types (namely strings and labels). String and labels expressions can only be assigned to a variable of the same type.

If a real or long real value is assigned to an integer or Boolean type variable, a rounding process occurs.

```
I := X
```

results in an integer value equal to `ENTIER(X + 0.5)` being assigned to `I`.

NOTE: in taxi, all assignments, expressions, conditions are at all effects numerical values. It depends on the contest if it is a condition (0 is false, any other value is true), or an assignment of an expression. Basing on this, a condition can be assigned as a numeric expression (this was illegal in ALGOL 60) or a value can act as a conditional. Consider the following examples:

```
I := R > 1
```

stores in `I` the value `-1` (default true condition) if `R` is greater than 1, or zero if not, while

```
IF R THEN PRINT(R)
```

prints `R` only if it is not zero, because `R` is evaluated as a condition (zero is false, not zero is true), and it does not need to be a Boolean value.

This is a minor deviation (an enhancement in effect) from ALGOL 60 and the ISO documents.

6.2.1. Multiple assignments

A value is assigned simultaneously to several variables of the same type by multiple assignments. This takes a form such as

```
P := R := S := X + Y - Z ;
```

where the result of adding `Y` to `X` and subtracting `Z` is assigned to `P`, `R`, and `S` simultaneously.

Note here that a multiple assignment with an equality test at the end is not ambiguous:

```
A := B := C := D = 0;
```

In languages using = as the assignment operator *and* the equality test operator, the `D = 0` part is ambiguous, because it can be the last item od the chain *or* the condition test. In ALGOL this is not possible, because the assignment operator and the equality test operator are different, and so the previous operation assigns to `A`, `B` and `C` the condition `D = 0`.

All identifiers on the left-hand side of multiple assignments don't need to be of the same type (but results may differ if integer and real variables are mixed).

6.2.2. Parenthesized assignments

A parenthesized assignment may be substituted for any operand in an expression. For example,

```
X := (Y := P+Q)/Z;
```

This causes the assignment to `X` to be made *after* the inner expression `Y := P+Q` is evaluated. Where a type conversion is performed as part of an embedded assignment, the operand type is the same as that assigned to the variable in the embedded assignment. Thus, if I is an integer and X is a real

```
X := (I := 3.4)
```

sets `I` equal to 3 and `X` equal to 3.0.

6.3. EVALUATION OF EXPRESSIONS

All expressions in taxi (as in the DECsystem-IO/20 ALGOL) are evaluated observing the normal algebraic rules of precedence, including bracketing.

The following Table 5-4 resumes the precedence levels of all the math and Boolean operators. The lower is the level, the lower is the precedence.

Table 5-4
Precedence

Precedence Level	Operator
10	+ − (prefix number signs) NOT ~
9	(...) parentheses (evaluated immediately)
8	^ **
7	* / \ ÷ DIV REM MOD
6	+ −
5	< <= = > >= #
4	AND /\
3	OR \/
2	IMP ->
1	EQV ==

Within the precedence structure, expressions are always evaluated from left to right. For example, if `X` is a scalar, and `F` a procedure (see Chapter 11) that alters `X`,

```
X := X + F ;
```

may have a different effect than

```
X := F + X
```

This is known as the *side effect* phenomenon.

Consider also:

```
A[I] := (I := I+1)
```

The subscript `I` is always evaluated before `I` is incremented, as it is to the left of the embedded assignment, within the statement. Thus the above expression is equivalent to

```
J := I;
I:= I+1;
A[J] := I;
```

The user can always predict the order of evaluation of an expression and can count on such things as

```
X := (P := P+Q)/(P+R)
```

being evaluated correctly, thus giving the same result as

```
P := P+Q ;
X := P/(P+R);
```

The final type of an expression depends on the types of the operands:

- if an expression involves only Booleans or Boolean procedures, the result is Boolean;
- if an expression involves Booleans or Boolean procedures and at least one Integer or Integer procedure, the result is Integer;
- if an expression involves Booleans or Boolean procedures or Integers or Integer procedure and at least one Real or Real procedure, the result is Real;
- if an expression involves Booleans or Boolean procedures or Integers or Integer procedures or Reals or Real procedures and at least one Long Real or Long Real procedure, the result is Long Real.

E.g.

```
PRINTLN(true*true);
PRINTLN(24*true);
PRINTLN(3.4&1*24*true);
PRINTLN(1&&1*3.4&1*24*true);
```

```
 true
          -24
-8.160000000&+2
-8.16000000000000014&&+3
```

6.4. COMPOUND STATEMENTS

A compound statement (or block) consists of a set of statements, preceded by `BEGIN`, separated by semicolons, and terminated by `END`. ALGOL statements, unlike those in FORTRAN, are terminated by a semicolon, not by the end of a line of text.

For example:

```
BEGIN
      I := 3; J:= 4;
      K := I + J;
      X := K
END
```

is a compound statement. Semicolons do not need to appear after the `BEGIN` or before the `END` (but their usage is not illegal); `BEGIN` and `END` act as a type of bracket (`END` is also a terminator).

The usefulness of compound statements will become apparent in later chapters.

7. CONTROL TRANSFERS, LABELS, AND CONDITIONAL STATEMENTS

7.1. LABELS

A label is a method of marking a place in a program so that control can be transferred to that point from elsewhere in the program.

taxi (as the DEC system-10/20 ALGOL) uses identifiers as labels. These identifiers are placed before statements and are followed by a colon.

A label name must begin with a letter or a number, and optionally be followed by one or more letters or decimal digits or dots. A label name, as an identifier, may not contain more than 64 characters.

Numeric labels are permitted, as established in the Revised Report (but were not implemented in DEC system-10/20 ALGOL).

For example:

```
COMP: X:= X + Y
```

is a statement labelled by `COMP`.

More than one label can be attached to a statement if required; thus,

```
LAB1:  2000:  Y:= 0 ;
```

NOTE: only digits, letters and the dot are allowed in labels; if you want to use more than one single word for your label, use the dot as a separator.

E.g.:

```
SQUARING OPERATIONS:        ! this label is illegal;

SQUARING_OPERATIONS:        ! this label is illegal;

SQUARING.OPERATIONS:        ! valid label;
```

I don't know what was the behaviour in DEC system-10/20 ALGOL or the ISO specifics, but I think anyway that this is a minor issue.

7.2. UNCONDITIONAL CONTROL TRANSFERS

A transfer of control, or "jump", to a statement in a program is handled by a `GOTO` statement. This statement consists of the word `GOTO` followed by the name of the label attached to the relevant statement. The two words `GO TO` can be used instead of the word `GOTO` in any statement where `GOTO` can be used. Thus:

```
LAB: I := J := 3;
K := I + J;
GOTO LAB;
```

is an example of a somewhat tedious program. To write any reasonable program, it is necessary to be able to jump conditionally.

7.3. CONDITIONAL STATEMENT

Conditional statements provide a method to make the execution of either a statement or a compound statement dependent on some condition in the program, such as the value of a variable. The simplest form of a

conditional statement is

```
IF B THEN S
```

where B is some Boolean expression, and S is a statement. For example:

```
IF X < 0 THEN I := I + 1
```

Here, X < 0 is the Boolean expression and I := I + 1 is the statement which is obeyed if and only if the Boolean condition is true, that is, if X is negative.

A more general form of a conditional statement is

```
IF B THEN S1 ELSE S2
```

In this case, the statement S1 is obeyed if and only if the Boolean expression B is true, and S2 is obeyed if and only if it is false. To eliminate the "dangling ELSE ambiguity" (a construction in which an ELSE could be paired with either of two THENs) and to avoid ambiguity in case of complex multiline statements, S1 and S2 should be only direct statements or assignments (for instance GOTO or A:=1) and not a complex statement like an IF conditional, a FOR, or a WHILE statement. In case you have to use such statements after THEN or ELSE, enclose them in a block (between a BEGIN and END).

Example:

```
BEGIN  INTEGER I;
   I := 0;
LAB:   I := I + 1;
       IF I < 100 THEN GOTO LAB
END
```

is a simple way of counting to one hundred.

More sophisticated methods are shown in Chapter 14.

8. CYCLE STATEMENTS

8.1. THE FOR STATEMENTS

The FOR statement enables the user to iterate a portion of the program in a fashion similar, to but more sophisticated than, FORTRAN's DO loop.

The general format is

```
FOR V := FORLIST DO S
```

where V is a variable and S is a statement (compound or simple).

FORLIST can consist of any number of FOR elements (separated by commas). A FOR element takes one of the following forms:

1. An expression:

```
E
```

2. A STEP-UNTIL element taking the form:

```
E1 STEP E2 UNTIL E3
```

3. A WHILE element taking the form:

```
E WHILE B
```

where B is some Boolean expression.

Any number of FOR elements may appear in a FOR statement, in which they are executed serially. Consider the following examples:

```
FOR I := 3,5,10 DO ...
FOR X := 2.5,5.0,10.0 DO ...
FOR J := 1,2,5 STEP 5 UNTIL 20 DO ...
```

8.1.1. The STEP-UNTIL element

This particular form deserves closer inspection. Consider

```
FOR I := B STEP I UNTIL N DO S
```

The statement S is obeyed with I taking an initial value of B, and being incremented by I units until the final value N is achieved. No cycle is performed if B, N and I cannot satisfy the surviving condition. (E.g. if B>N and I are positive.)

taxi (as the DEC system-10/20 ALGOL) allows the abbreviated form

```
FOR V := E1 UNTIL E3 DO S
```

instead of

```
FOR V := E1 STEP 1 UNTIL E3 DO S
```

The value of the running index V, at exit, yields the last useful value of the cycle. (Unlike C, for instance, that sets this running index to the first invalid value not included in the cycle.)

8.1.2. The WHILE element

A FOR statement with a single WHILE element takes the form

```
FOR V := E WHILE B DO S
```

This is interpreted as follows:

```
L1:     V := E;
        IF NOT B THEN GOTO L2;
        S;
        GOTO L1;
L2:
```

Once again, the complexity of the loop may be affected by changing V and E within the loop.

8.2. THE WHILE STATEMENT

The WHILE statement is an enhancement of ALGOL 60 provided in DEC system-10/20 ALGOL (not in the ISO 1538-1984) and adopted by taxi too. The general form of the statement is

```
WHILE B DO S
```

and is interpreted as follows:

```
L1:     IF NOT B THEN GOTO L2;
        S;
        GOTO L1;
L2:
```

8.3. GENERAL NOTES

The following notes must be taken into account when using the FOR and WHILE statements.

1. Within a FOR statement of any kind, the user can change the controlling variable or any other variable appearing within the action of the loop. Such changes predictably affect the execution of the loop by the rules given above.

2. On exit from a FOR statement either by jumping out of the loop or by exhausting the FOR elements, the controlling variable has a well-defined value equal to the last assigned value of the controlling variable. This may not be true of other ALGOL 60 implementations depending on Section 4.6.5 of the Revised Report, where the following established rule is reported: "*If the exit is due to exhaustion of the* FOR *list, on the other hand, the value of the controlled variable is undefined after the exit*."

9. ARRAYS

9.1. GENERAL

Arrays are essentially lists of variables of the same type, allowing the user to address each variable individually using a common name and a unique subscript or subscripts. In the simplest case, an array is a vector and is known as a one-dimensional array. A matrix is a two-dimensional array, etc.

There is no limit to the number of subscripts allowed, other than those imposed by the ability of the computer to store the array.

9.2. ARRAY DECLARATIONS

Arrays may be of type integer, real, long real, Boolean, or string and these are declared similarly to scalar variables, except the size of the array must be stated. For each subscript that the array possesses, a lower and an upper bound called the "bound pair" for that subscript must be given.

For example, to declare two one-dimensional integer arrays `A` and `B` with lower bound 1 and upper bound 5:

```
INTEGER ARRAY A,B[1:5]
```

NOTE: The lower and upper bounds must be enclosed in square brackets and separated by a colon.

When there are two or more subscripts, the declaration is similar, and the bound pairs are separated by commas. Thus

```
LONG REAL ARRAY P,Q,R[-5:2,0:10]
```

declares three long real arrays, `P`, `Q` and `R`, with the first subscript bounded by -5 and 2 and the second subscript bounded by 0 and 10.

Arrays of the same type but of different sizes may be declared in the same statement:

```
REAL ARRAY A[1:10], B,C[1:10,1:12]
```

NOTE: In the case of real arrays, the `REAL` may be omitted in the declaration, and is assumed by default, thus:

```
ARRAY A[1:10], B,C[1:10,1:12,-5:5]
```

behaves like `REAL ARRAY`...

The bounds in an array need not be static, as in the examples above, but can be any arithmetic expressions, which are evaluated to give an integer value for the individual bound pairs. The use of such dynamic array declarations will become apparent later. The bound limit should not exceed 2147483647 in magnitude, but in general memory problems should arise with much lower bounds (unless your computer has a ***very huge*** memory).

NOTE: arrays dimensions may be more than two... if you imagine how to use such multi-dimension arrays.

NOTE: string arrays elements size is fixed, set by default to 255, and cannot be changed.

NOTE: to declare a real array `A` and a real variable `V`, one could be tempted to write:

```
REAL ARRAY A[1:10], V;         ! invalid form;
```

Unfortunately, the ARRAY keyword (and the specifier that precedes it, for example REAL) form a unique statement; thus, two separate instructions must be used, i.e.

```
REAL ARRAY A[1:10]; REAL V;        ! valid form;
```

9.3. ARRAY ELEMENTS

An individual element of an array can be referred to by following the name of the array by a list of subscripts in square brackets. The number of subscripts must be identical to the number in the array declaration. Thus, a typical element of A used in the last declaration might be

```
A[5] or A[9] or generally, A[I]
```

where I is some integer expression or, in general, any expression whatsoever, with the limitation that its value, when used as a subscript and evaluated as an integer, is in the range of that array A dimension.

As an example of the use of arrays, consider the declaration

```
REAL ARRAY D,E,F [1:10,1:10]
```

and suppose the operation required, was to set F equal to the matrix product of D and E:

```
FOR I := 1 UNTIL 10 DO
FOR J := 1 UNTIL 10 DO
BEGIN X := 0;
   FOR K := 1 UNTIL 10 DO X := X + D[I,K]*E[K,J];
   F[I,J] := X
END
```

NOTE

1. In the above example, X is used to accumulate the Inner product of the multiplication for all values of I and J. The variable X was used instead of F to facilitate the computation.

2. An element of an array of a particular type may be used anywhere that a scalar variable of the same type may be used, that is in every possible math expression, but not as the controlling variable in a FOR statement. This is a minor deviation from the DEC system-10/20 ALGOL.

10. BLOCK STRUCTURE

10.1. GENERAL

An ALGOL program structure is somewhat more complicated than other high-level languages, such as FORTRAN. An ALGOL program consists of a series of "blocks" arranged hierarchically. A block consists of the words `BEGIN` and `END` enclosing the declarations and (optionally) statements.

Thus:

```
BEGIN
   ...
   BEGIN
      ...
   END;
   ...
   BEGIN
      ...
      BEGIN
         ...
      END;
      ...
   END;
   ...
END.
```

is an ALGOL program, assuming appropriate declarations and statements in the blocks. Notice the semicolons at the end of each inner block and the dot at the end of the outermost block.

The block structure offers the user many interesting features not available in non-block structured languages. For instance, the user may declare an identifier that appears to conflict with another identifier in an enclosing block. Thus:

```
BEGIN INTEGER I;
   ...
   BEGIN INTEGER I;
      ...
   END
   ...
END
```

There is no conflict, here, because they are two different `I`s. The only `I` that statements in the outer block can "see" is the one in the outer block. Similarly, any statement in the inner block will always use the I in that block. Such a declaration in an inner block is known as a "local" variable and takes precedence over declarations occurring at an outer or more "global" level. In general, all variables can be "seen" from any point in a program that is either in the same block as the declaration or in a block that is enclosed by the block in which the declaration of the variable occurred. Note that a more local variable is always taken in preference to a relatively global variable. Consider the following example:

```
BEGIN INTEGER I, J;
   [1]
   BEGIN INTEGER J,K
      [2]
   END;
   BEGIN INTEGER I,K
```

```
        [3]
    END
END
```

Any statements occurring at point [1] can see the declarations of I and J, which are local but cannot see the declarations of J and K in the first inner block, or the declarations of I and K in the second inner block. At [2], the local variables J and K can be seen, as can the global variable I in the outer block. The global variable J is not seen because the local variable J takes precedence over it; the variables I and K in the second inner block are not seen at all. A similar situation occurs at [3]; here both local variables I and K, as well as the global variable J, are seen.

Note that the "scope" of a variable is the set of all places in a program where it can be seen and therefore used. This term will be used frequently throughout this text.

In general, local variables are more efficient to use than global ones. This statement is also true of most ALGOL 60 implementations. Where a global variable is used frequently, a local variable should be assigned as having the same value and used instead. For example:

```
BEGIN INTEGER I;
    ...
    I := ...
    BEGIN INTEGER II;
        II := I;
        ...
        ... II ...
    END
    ...
END
```

Here, in the inner block, a local variable II is used and assigned the value of the global variable I for use throughout the local block.

10.2. ARRAYS WITH DYNAMIC BOUNDS

The concept of the scope of a variable can be applied most usefully to arrays. In taxi (and in the DEC system-10/20 ALGOL), all arrays are constructed at execution time (that is, no fixed space is reserved during compile-time), irrespective of whether their bounds are static or dynamic. When a declaration of an array is encountered within a block, the space required to construct it is obtained and the array is laid out. When the end of the block enclosing the array is reached, that is, the array variable is no longer within scope, the space utilized by the array is recovered and can be used later for other arrays.

Consider the case of a problem in which the size of an array to be used in a calculation is dependent on the data to be processed. The programmer has the choice of making the array large enough to cope with the worst of the cases or constructing the array with dynamic bounds to suit the size required by the particular data. The first method has the disadvantage of wasting space on many occasions. The latter method only has the minor disadvantage of the overhead needed to construct the array. Such overhead is very small compared to the running time of most programs, therefore, the second method is more desirable.

Consider the following example:

```
BEGIN INTEGER N;
L: N := ...
    ...
    BEGIN ARRAY A[1:N,1:N];
        ...
```

```
    END;
    GOTO L
END
```

A value for N is calculated in this example, possibly dependent on some data read into the program, and used to declare the array A, which is used to process the data in the inner block. When the end of the inner block is reached, the space used by A is recovered and control passes to L, where another value for N is calculated, and the process repeated.

11. PROCEDURES

Procedures are similar in concept to the FORTRAN subroutines, but with more sophisticated and general applications.

A *procedure* is a portion of an ALGOL program that is given a name for identification and can be "called" from any part of a program which is in the scope of the body of the procedure. A procedure can execute a series of statements, and can eventually return a value to the procedure body. In addition, it may or may not have parameters. User procedure names are called in case-sensitive mode, so that if you define procedure `TEST`, you have to call it as `TEST`, because `test` would not be found; this is in accordance with the user variables rules for what about the fictitious variables, that must be created in case-sensitive mode. (In the above examples, the fictitious associated variable would be `TEST`, which is different from `Test` or `test`.)

A *formal parameter* is an entity declared and used in the body of a procedure; an *actual parameter* is the value passed to a formal parameter in the calling environment (it may be any legal string or numeric expression).

In taxi (as in the DEC system-10/20 ALGOL), a procedure can be one of the following types: integer, real, long real, Boolean, string, label or typeless. The formal parameters of a procedure (known as "dummy variables" in FORTRAN), can be one of the following types: integer, real, long real, boolean, string, label as scalars; they can be values, arrays, or procedures. There are nineteen different types of formal parameters. (See table 11-1.)

Whenever the formal parameter associated with the actual parameter in a procedure body appears in the body of the procedure, the actual parameter is re-evaluated as if it appeared in the procedure body at that point. For example, if the actual parameter is an array element like `A[I]`, the element is re-evaluated using the current value of `I` (i.e., the value available at each time the formal parameter is used, not at the time the procedure body is entered).

11.1. PARAMETERS CALLED BY "VALUE"

Calling parameters by *value* is the most common and, except for arrays and strings, the most efficient way to pass a parameter to a procedure. The value of the expression presented in a procedure call, known as the actual parameter, is evaluated on entry to the procedure and assigned to a formal parameter within the procedure. This formal parameter acts like a local variable in the procedure which is initialized, the initial value being that of the actual parameter supplied in the call to the procedure.

In the case of arrays or strings, since a new copy of the array or string is made, this type of parameter-passing for arrays and strings should be avoided unless specifically required.

Parameters called by value are specified in the `VALUE` list.

NOTE: labels are always passed by value, independently from the presence of the statement VALUE (which is optional).

11.2. PARAMETERS CALLED BY "NAME"

Calling parameters by "name" is a useful method of passing a parameter to an ALGOL procedure. Whenever a variable or an array item is passed to a parameter by name, the parameter acts like a substitute of the variable or the array item, that will inherit all the changes applied to the parameter.

Table 11-1 shows the different types of formal parameters, with valid actual parameters that can be substituted in a procedure call.

Table 11-1
Parameters in a Procedure Call

Formal Type	Actual Parameter
Integer Real Long Real Boolean	Any arithmetic expression
String	Any string expression (refer to Chapter 13)
Label	A label or switch element (refer specially to Chapter 7 and Chapter 12)
Switch	A switch
Integer Array	An array of type integer*
Real Array (or Array)	An array of type real*
Long Real Array	An array of type long real*
Boolean Array	An array of type Boolean*
String Array	An array of type string
Procedure	A non-type procedure
Integer Procedure Real Procedure Long Real Procedure Boolean Procedure	A procedure of type integer, real, or long real or Boolean
String Procedure	A procedure of type string
Label Procedure	A procedure of type label

(*) In the case where the array parameter is called by value, any arithmetic type (integer, real or long real) array is allowed as an actual parameter. A conversion of the type takes place during the copying process.

You may wonder why there's not the Label array case; this case is performed by the SWITCH declaring procedure, because a switch is a sort of string array with labeling properties. (See SWITCH.)

11.3. PROCEDURE HEADINGS

Procedure headings identify the type of procedure, the number and the type of parameters. A procedure heading consists of:

1. The type of procedure (omitted in the case of typeless procedures).

2. The word PROCEDURE followed by the name of the procedure.

3. A semicolon if the procedure has no parameters, otherwise

4. A list of the formal parameters, enclosed in parentheses, separated by commas, and followed by a semicolon.

5. Specifications of the formal parameters.

A procedure identifier must begin with a letter, and optionally be followed by one or more letters, decimal digits or dots. A procedure identifier may not contain more than 64 characters.

Omitting formal parameter specifications, this looks like

```
LONG REAL PROCEDURE LR;
BOOLEAN PROCEDURE BOOLE(I,J,K);
PROCEDURE CALC(THETA,X);
```

The formal parameter specification that follows consists of a list of descriptions of the formal parameters, appearing in any order, and a value specification if any of the parameters are to be called by value. (If this is omitted, the parameters, by default, will be called by name.) For example, the specification of the formal parameters for the second example above might be:

```
VALUE I,J; INTEGER I,J,K;
```

meaning that all three formal parameters are of type integer (scalars), and `I` and `J` are to be called by value, while `K` is to be called by name. A typical formal parameter specification for the third example might be:

```
REAL PROCEDURE THETA; ARRAY X;
```

NOTE: Procedure headings must precede the body of the procedure.

NOTE: In a procedure heading spread in two or more lines (for instance the `PROCEDURE name()` line and the variable declaration lines), comments should be avoided, though permitted, because they obfuscate the specifications section. So that instead of the erroneous

```
REAL PROCEDURE ALPHA(T,H);
COMMENT FOR ALPHA IN A BAD AREA! ;
INTEGER T; REAL H; BEGIN ...
```

use the following form

```
COMMENT FOR ALPHA IN A GOOD AREA! ;
REAL PROCEDURE ALPHA(T,H);
INTEGER T; REAL H; BEGIN ...
```

which is more clear. Anyway, follow your sense.

11.4. PROCEDURE BODIES

The body of a procedure is that part which follows the procedure heading, and consists of a single statement, a compound statement, or a block. In the last-mentioned case, there may be declarations of local variables within the block, and also other blocks or procedures. Consider the following examples of realistic procedures:

1. A real procedure, `SQUAREROOT`, to calculate the square root of a real quantity. The first parameter is the quantity, the second is a label that is used as an escape if the quantity is found to be negative. The result of the procedure is the square root of the quantity. Note how the result of the calculation is assigned to the procedure by placing the name of the procedure on the left-hand side of an assignment.

```
REAL PROCEDURE SQUAREROOT(X,L);
       VALUE X; REAL X; LABEL L;

BEGIN  REAL Y,Z;
       IF X < 0 THEN GOTO L;
       Y := (1+X)/2;
```

```
IT:     Z := (X/Y + Y)/2;
        IF ABS(Z-Y) < 1&-6 THEN GOTO OK;
        Y := Z;  GOTO IT;

OK:     SQUAREROOT := Z;

EXIT:   END SQUAREROOT;
```

The previous example uses the Newton-Raphson method for finding the square root of a number by taking an initial approximation `(1 + X)/2` and iterating until the difference between successive approximations is less then 1&-6. The procedure is again described below, with the aid of some commentary. The DEC system-10/20 ALGOL alternative method of commentary (refer to 2.4) is used for brevity:

```
REAL PROCEDURE SQUAREROOT(X,L);
        VALUE X; REAL X; LABEL L;

        BEGIN  ! CALCULATES THE VALUE OF SQRT(X)
                 USING THE NEWTON-RAPHSON METHOD.
                 L IS USED FOR AN ESCAPE IF X < 0;

        REAL Y,Z;
        IF X < 0 THEN GOTO L;     ! EXIT IF X < 0;
        Y := (1+X)/2;             ! FIRST APPROXIMATION;

IT:     Z := (X/Y + Y)/2;         ! ITERATE;
        IF ABS(Z-Y) < 1&-6
             THEN GOTO OK;        ! TEST FOR CONVERGENCE;
        Y := Z;  GOTO IT;         ! OTHERWISE CONTINUE;

OK:     SQUAREROOT := Z;          ! FINAL RESULT;

        END SQUAREROOT;
```

2. A procedure `SUM` to sum of the values of any real procedure `G` over the integers `1 ..... N` (that is, iterating `N` times) where `N` is also a parameter of the procedure.

```
REAL PROCEDURE SUM(G,N);

        VALUE N; REAL PROCEDURE G; INTEGER N;

BEGIN   INTEGER I; REAL X;

         X := 0;

         FOR I : = 1 UNTIL N DO X := X + G(N) ;

         SUM : = X
END
```

NOTE: In this example, the formal parameter `G` is invoked so that the actual procedure substituted for `G` is called. It's the user responsibility to call `G` with the same number of parameters with the same types and order required by the procedure passed as the argument, otherwise, an error is raised.

11.5. PROCEDURE CALLS

In the preceding example, procedure G was *called*. Since G is a function procedure, it is only necessary for its name to appear in an expression for the procedure to be entered with the actual parameters specified substituted for the formal parameters.

The function procedure SQUAREROOT of a preceding example can also appear in an expression, for example:

```
P := SQUAREROOT(Z+0.5, ERR)
```

causes the square root of Z + 0.5 to be calculated.

An example of the use of the procedure SUM can be used to calculate the sums of the square roots of the first J integers, with the result squared, as follows:

```
X := SUM(SQUAREROOT,J)^2;
```

Here is a further example of a procedure and the calls:

```
PROCEDURE MATRIXMULT(A,B,C,N);

VALUE N; ARRAY A,B,C; INTEGER N;

  BEGIN   INTEGER I,J,K; REAL X;

  COMMENT THIS PROCEDURE PERFORMS THE MATRIX
  MULTIPLICATION OF 'B' AND 'C' AND STORES THE RESULT
  IN 'A'. THE ARRAYS ARE ASSUMED TO BE SQUARE
  AND OF BOUNDS 1:N,1:N;

  FOR I := 1 UNTIL N DO
  FOR J := 1 UNTIL N DO

  BEGIN X := 0;

  FOR K := 1 UNTIL N DO X := X +
      B [I,K]*C[K,J];
  A[I,J] := X

  END
END;
```

A typical call for this procedure might be

```
MATRIXMULT(E,F,G,N) ;
```

or

```
MATRIXMULT(E,F,F,N) ;
```

Since the arrays are called by name, a call such as MATRIXMULT(E,E,F,N); would give rather interesting results.

This call could be made to work by calling B and C of MATRIXMULT(A,B,C,N); by value. However,

this would increase the overhead of the procedure considerably.

11.6. PROCEDURE SCOPES

As shown in the ISO 1538-1984, ALGOL lets design nested procedures, up to any level. Let's consider the following example:

```
BEGIN

   PROCEDURE PROC1(...);       BEGIN
   ....

      PROCEDURE PROC3(...);          BEGIN
      ....

         PROCEDURE PROC5(...);             BEGIN
         ....
         END PROC5;

      ....
      END PROC3;

   ....
   END PROC1;


   PROCEDURE PROC2(...);       BEGIN
   ....

      PROCEDURE PROC4(...);          BEGIN
      ....

      END PROC4;

  PROCEDURE PROC6(...);          BEGIN
  ....

  END PROC6;

   ....
   END PROC2;

....
END.
```

If level 0 is assumed to be the level of the most outer block (root block), the example illustrates the creation of `PROC1` at level 0, which creates `PROC3` at level 1, which creates `PROC5` at level 2. Then follows the creation of `PROC2` at level 0, which creates `PROC4` and `PROC6` at level 1.

In general, the following law is respected: a procedure may be called if created at level 0 or in the same level of the caller; a procedure can thus call itself.

Here are the scopes of the procedures in the example:

1. root block sees PROC1 and PROC2;

2. PROC1 sees PROC3 (created in its space) and PROC2 (at the same level).

3. PROC3 sees PROC5 (created in its space) and PROC1/PROC2 (at level 0).

4. PROC5 sees PROC1 and PROC2 (at level 0).

5. PROC2 sees PROC4 and PROC6 (created in its space) and PROC1 (at level 0).

6. PROC4 sees PROC6 (at the same level) and PROC1/PROC2 (at level 0).

7. PROC6 sees PROC4 (at the same level) and PROC1/PROC2 (at level 0).

8. All procedures see themselves.

11.7. ADVANCED USE OF PROCEDURES

11.7.1. More on argument passing by name

We have seen in previous sections that a call by name with a variable argument binds the two variables (the one passed and the one in the body of the procedure) so that an assignment to the latter is also an assignment of the former.

If instead any *formula* is passed to a reference argument (that is a number, an expression, a function, etc.), the argument is evaluated at run time returning its proper value. This is legal as far as the argument, in the body of the procedure, is used only in the right part of an assignment (that is, in an expression), because the number is legally admitted as an element of an expression.

If instead this argument is used in the left part, that is an assignment to it is tried, an error is raised, because there is no variable associated. Besides, since the calculated value is always referred to the original call, any instantiation would be reset to the value originally passed at each invocation.

See the following example:

```
BEGIN
  REAL R;
  PROCEDURE TEST(X);
  BEGIN
     X:=3; PRINTLN(X);
  END;

  R:=4;
  TEST(R);
  PRINTLN(R);
  TEST(5*4);
  PRINTLN(R);
END.
```

The first call to the TEST procedure links the reference of R to X, and when X is assigned 3, this value is assigned also to R; the second call to TEST passes a numeric expression, which is inherited by X, but when the assignment is tried, an error is raised. The total output is of course:

```
 3.000000000
 3.000000000

taxi runtime processor error in procedure test,  at line 5:
Illegal assignment due to invalid lefthand type.
X:=3;
^
Error code 88
```

Let's now suppose, absurdly, that the assignment would be possible. The argument `X` had inherited the value 20 (that is 5*4) in the creation of the procedure instance and the assignment would try to set `X=3`; the assignment fails in reality, because there is no linked variables, but we have supposed that it succeeds. So, when `X` is printed, since there was no assignment, it would recall it's former attribution stored in it (remember that is was not passed by value). So that it would print 20 instead of 3. To avoid these inconsistencies taxi (as many other compilers/interpreters) bans such assignments, and returns an error message.

Anytime you have to alter a variable in a procedure, you must assure either to create it as a procedure local variable or to use the `VALUE` specifier while creating the procedure. In this case, the `VALUE` specifier would simply create a normal variable with the original value passed (20 in the example) that any further assignment would reset to the new assigned value.

Another interesting consequence is this; if you plan to use a string by name in a procedure, you may think that using it in the right part of an assignment as a char subscript is legal and painless; for instance:

```
INTEGER I,J;
PROCEDURE TEST(X) LETTERS:(L) FIRST:(C);
STRING X; INTEGER L,C;
BEGIN
   L:=LENGTH(X);
   C:=X.[1];
END;

TEST("TODAY IS A GOOD DAY",I,J)
```

The previous piece of code will fail while assigning `C:=X.[1]`, because the search of the first character of the string is made upon the referenced string, and not in the body of `X`, but there is no reference string associated, because `X` was feed with a literal string. The result is the error message of not found string:

```
taxi runtime processor error in procedure test,  at line 10:
Illegal number, undeclared numeric variable or undefined numeric func-
tion.
C:=X.[1];
         ^
Error code 81
```

In these cases, either you associate a string `S:="TODAY IS A GOOD DAY";` and pass `S` to `TEST` or, since you never have to instantiate `X`, you set it as `VALUE`. In both cases, you will get the correct answer.

11.7.2. Jensen's Device

This method of using a procedure exploits the power and flexibility of the call-by-name concept. Consider the following example:

```
REAL PROCEDURE SUM(I,N,X); VALUE N; INTEGER I,N; REAL X;
BEGIN   REAL Y;
```

```
            Y:=0;
            FOR I:=1 UNTIL N DO Y:=Y+X;
            SUM:=Y;
      END SUM;
```

On the surface, the procedure appears to calculate the value of `N * X`. However, consider the call

```
Z := SUM(J,10,A[J]);
```

and remember that `J` and `A[J]` are parameters called by name. Since `I` and consequently `J` take new values, each `X` in the loop is evaluated as a particular value of `A[J]`, using the value of `J` just assigned. Hence the above call calculates

```
A[1] + A[2] + ..... + A[10].
```

Similarly, the call

```
Z := SUM(K,M,A[I,K]*B[K,J]);
```

calculates the `(I,J)` th inner product of `A` and `B`.

The following listing shows a complete program in which Jensen's device is used:

```
BEGIN
   INTEGER L; L:=10;
   REAL ARRAY ARR[1:L];
   INTEGER K; REAL RES;

   COMMENT: JENSEN'S DEVICE TEST;
   REAL PROCEDURE SUM(I,N,X); VALUE N; INTEGER I,N; REAL X;
   BEGIN   REAL Y;
      Y:=0;
      FOR I:=1 UNTIL N DO Y:=Y+X;
      SUM:=Y;
   END SUM;

   COMMENT MAIN;
   FOR K:=1 UNTIL L DO ARR[K]:=K;
   RES := SUM(K,L,ARR[K]);
   PRINT (RES,5,7)
END MAIN.
```

This program prints:

```
      55.0000000
```

which is precisely the sum of `A[1] + A[2] + ... + A[10]`, where `A[J]=J`.

11.7.3. General Problem Solver (GPS)

Another technique can be illustrated to show the power of the calling by name feature in ALGOL. This was introduced by D.E. Knuth and J.N. Merner in their article titled "ALGOL 60 confidential"[8]. The algorithm is simple:

```
        REAL PROCEDURE GPS(I1,N,Z,V);
        REAL I1,N,Z,V;
        BEGIN
                FOR I1 := 1 STEP 1 UNTIL N DO Z := V;
                GPS := 1
        END;
```

This algorithm was later exposed by B. Randell and L.J. Russell in their 1964 book "Algol 60 implementation"[9], the best text about how to implement an ALGOL compiler[10] called Whetstone ALGOL. This is the book I consulted. Let us Randell and Russell speak (page 253): "*The single example chosen to illustrate the workings of the Whetstone ALGOL Compiler is based on the procedure GPS (General Problem Solver) in the article* ALGOL 60 Confidential *by Knuth and Merner. In this article the possibilities of recursive procedures and parameters called by name are demonstrated by using GPS in several assignment statements, one of which has the effect of multiplying two matrices together, and which is presented as a challenge to compiler writers. This use of the procedure GPS, which is rightly described by Knuth and Merner as* just ALGOL for ALGOL'S sake*, nevertheless demonstrates several fundamental features of ALGOL which are of great practical value, and hence is worth using to demonstrate translation techniques. In order to reduce the task of description to more manageable proportions, the working of the Whetstone Compiler is demonstrated on a program using GPS to set up a matrix* `A`*, in which the element* `A[i,j]` *has the value* `i+j`. [...] *Though it is not claimed that the method of implementation is in any way optimum, the example shows that the standard mechanisms of the Whetstone Compiler for handling recursive procedures and parameters called by name can easily deal with the apparent complications of General Problem Solver.*

The following listing shows a complete program in which the GPS device is used:

```
BEGIN
  REAL I;
  REAL J;
  REAL ARRAY A[1:4, 1:5];

  REAL PROCEDURE GPS(I1,N,Z,V);
     REAL I1,N,Z,V;
     BEGIN
        FOR I1 := 1 STEP 1 UNTIL N DO Z := V;
        GPS := 1
     END;

  COMMENT MAIN SECTION;
  I := GPS(J, 5.0, I, GPS(I, 4.0, A[I,J], I + J));

  COMMENT OUTPUT OF I AND J;
  OUTREAL(1,I);OUTSTRING(1,"\n");
  OUTREAL(1,J);OUTSTRING(1,"\n");

  COMMENT OUTPUT OF THE ARRAY;
  FOR I:=1 STEP 1 UNTIL 4 DO BEGIN
     FOR J:=1 STEP 1 UNTIL 5 DO BEGIN
        OUTREAL(1,A[I,J]);
     END;
```

[8] Published in Commun. ACM 4, 6 (June 1961), pages 268-272.

[9] Published by the Automatic Programming Information Centre (APIC), Brighton College of Technology, England, 1964, Academic Press, London and New York.

[10] Just to clear the field, I built taxi following my personal model, not using the techniques exposed in this book. And this can be derived by the fact the Whetstone Compiler is far more clever than mine.

```
      OUTSTRING(1,"\n");
   END;
 END.
```

The program is built for a general ISO ALGOL, avoiding all the DEC features or new procedures that taxi has builtin. As such, this program can be executed on a wider range of ALGOL compilers/interpreters. This program is here reported with a 4x5 matrix (the dimensions in the book were 2x3, supposedly to ease tracking down the program flow, but the principle remains the same and you can change the dimensions at will).

The output of the program follows:

```
  1.000000000
  5.000000000
  2.000000000  3.000000000  4.000000000  5.000000000  6.000000000
  3.000000000  4.000000000  5.000000000  6.000000000  7.000000000
  4.000000000  5.000000000  6.000000000  7.000000000  8.000000000
  5.000000000  6.000000000  7.000000000  8.000000000  9.000000000
```

The core of the procedure is of course the assignment `Z := V` (occurring in the second call to `GPS`), with `Z` bound to `A[I,J]` and `V` bound to `I + J`. (You can find the original program in the `files/` directory of the installation package as `gps.alg`.)

A far more complex example is also given in the book (page 266), conceived by Knuth and Merner themselves. This program calculated the product of two matrices using the GPS. The example given in the book is here copied and integrated in a full working program:

```
 BEGIN
   REAL I;
   REAL J;
   REAL K;
   COMMENT M = 4, N = 5, P = 3;
   REAL ARRAY A[1:4, 1:5], B[1:5, 1:3], C[1:4, 1:3];

   REAL PROCEDURE GPS(I1,N,Z,V);
   REAL I1,N,Z,V;
   BEGIN
      FOR I1 := 1 STEP 1 UNTIL N DO Z := V;
      GPS := 1
   END;

   COMMENT FILL UP MATRICES A AND B;
   FOR I:=1 STEP 1 UNTIL 4 DO
      FOR J:=1 STEP 1 UNTIL 5 DO
         A[I,J]:=I*J;
   FOR I:=1 STEP 1 UNTIL 5 DO
      FOR J:=1 STEP 1 UNTIL 3 DO
         B[I,J]:=I*J;

   COMMENT MAIN SECTION;
   I:=GPS(I, 1.0, C[1,1], 0.0) * GPS(I, (4-1) * GPS(J,(3-1) * GPS(K,5,
C[I,J],
   C[I, J]+A[I,K]*B[K,J]), C[I,J+1], 0.0), C[I+1,1], 0.0);

   COMMENT OUTPUT OF I AND J;
```

```
  OUTREAL(1,I);OUTSTRING(1,"\n");
  OUTREAL(1,J);OUTSTRING(1,"\n");
  OUTREAL(1,K);OUTSTRING(1,"\n");

  COMMENT
  OUTPUT OF THE ARRAYS;
  OUTSTRING(1,"ARRAY A IS\n");
  FOR I:=1 STEP 1 UNTIL 4 DO BEGIN
     FOR J:=1 STEP 1 UNTIL 5 DO BEGIN
        OUTREAL(1,A[I,J]);
     END;
     OUTSTRING(1,"\n");
  END;
  OUTSTRING(1,"ARRAY B IS\n");
  FOR I:=1 STEP 1 UNTIL 5 DO BEGIN
     FOR J:=1 STEP 1 UNTIL 3 DO BEGIN
        OUTREAL(1,B[I,J]);
     END;
     OUTSTRING(1,"\n");
  END;
  OUTSTRING(1,"\nMULTIPLICATION RESULT ARRAY C IS\n");
  FOR I:=1 STEP 1 UNTIL 4 DO BEGIN
     FOR J:=1 STEP 1 UNTIL 3 DO BEGIN
        OUTREAL(1,C[I,J]);
     END;
     OUTSTRING(1,"\n");
  END; END.
```

Again, the program is built for a general ISO ALGOL. This program is here reported with a 4x5 matrix `A`, with a 5x3 matrix `B` and of course with a 4x3 result matrix `C`. (You can find the original program in the `files/` directory of the installation package as `mgps.alg`.)

The output of the program follows:

```
  1.000000000
  2.000000000
  5.000000000
 Array A is
  1.000000000  2.000000000  3.000000000  4.000000000  5.000000000
  2.000000000  4.000000000  6.000000000  8.000000000  10.00000000
  3.000000000  6.000000000  9.000000000  12.00000000  15.00000000
  4.000000000  8.000000000  12.00000000  16.00000000  20.00000000
 Array B is
  1.000000000  2.000000000  3.000000000
  2.000000000  4.000000000  6.000000000
  3.000000000  6.000000000  9.000000000
  4.000000000  8.000000000  12.00000000
  5.000000000  10.00000000  15.00000000

 Multiplication array C is
  55.00000000  110.0000000  165.0000000
  110.0000000  220.0000000  330.0000000
  165.0000000  330.0000000  495.0000000
  220.0000000  440.0000000  660.0000000
```

A web page (an email, actually) where the GPS is also confidentially treated can be found at:

https://cseweb.ucsd.edu/~goguen/courses/230w02/GPS.html

The author is Dana Dahlstrom. She introduces the GPS in few words and presents some algorithms, the second being the latter here exposed. Towards the end, she adds: "*In fact, using GPS we can actually compute any computable function* [...] *using a single ALGOL assignment statement.*"

This is much more than a compliment for ALGOL!

11.7.4. Recursion

ALGOL procedures are recursive, that is, they may call themselves, directly or indirectly, to any reasonable depth. (The only restriction is the amount of core storage available to the object program.) An often-quoted and very inefficient method of calculating the factorial function of a small positive integer `N` is:

```
INTEGER PROCEDURE FACTORIAL(N);
VALUE N; INTEGER N;
      IF N = 1 THEN FACTORIAL := 1
      ELSE FACTORIAL := N * FACTORIAL(N-l);
```

This procedure has only one single statement and no local variables, and can, therefore, be written in a compact form. A call such as

```
J := FACTORIAL(6);
```

causes the procedure to be entered with `N` equal to 6. The call to `FACTORIAL` inside `FACTORIAL` enters the procedure a second time with `N` equal to 5, but this `N` is different from the one to the previous `N`, which retains its value of 6, and is stored in a different space. In this particular case, `FACTORIAL` is entered six times, the last time with `N` equal to 1.

NOTE: you'll be tempted (as I did) to see how far (and how inefficient) is this method. This depends of course on your machine, but when you reach the limit you will get a memory error.

Consider the following program:

```
BEGIN
   INTEGER PROCEDURE FACTORIAL(N);
   VALUE N; INTEGER N;
      IF N=1 THEN FACTORIAL := 1
      ELSE FACTORIAL:= N * FACTORIAL(N-1);

      PRINT(FACTORIAL(10));
END.
```

This program can be used with any positive value, trying to exploit the machine capabilities. For what about mine, a laptop with 4 Intel Core i3-5005U CPU at 2.00GHz and Linux OpenSuse Leap 15.1 as Operating System, these limits are reached:

- values from 1 to 12 return a meaningful value
- values from 13 to 49450 return always 2147483647
- values from 49451 on print "Out of Memory"

The first range is OK. The second range returns an infinitive integer representing the INTEGER type

proper limit. The third range means that if I use a value greater than 49450, the number of "suspended" processes waiting for the previous to complete are too much for my computer. Probably yours is much more clever...

11.7.5. Break and continue

Modern programming involves two procedures that are missing in ALGOL `BREAK` and `CONTINUE`.

The procedure `BREAK` is used to break a cycle (definite or indefinite) in some specific position and under specific conditions; one example in C is:

```
while (i<limit) {
        d+=3*i;
        if (d>=t) break;
        i++;
}
```

The same task in ALGOL cannot be replicated as-is, because `BREAK` is missing; but this situation can be completely emulated by the sage usage of `GOTO` and labels, for instance:

```
WHILE i<limit DO BEGIN
       d:=d+3*i;
       IF d>=t THEN GOTO rec;
       i:=i+1;
END;
rec:  comment continue here...;
```

taxi takes care, in case of a `GOTO` outside the cycle itself, of *dismissing* the cycle from memory, that is resetting the inner references of the cycle as if the cycle was never executed.

Analogously, `CONTINUE` is a procedure that resumes the cycle starting over again. This is useful when reading characters from a source; example in C:

```
while ((c=fgetc(fd)) {
       if (c=='A') continue;
       putchar(c);
}
```

This piece of source continues reading from the file source `fd`, printing all characters found but the capital `'A'`.

The same task in ALGOL is in DEC flavour:

```
redo:
WHILE (INSYMBOL(c)) DO BEGIN
       IF c=65 THEN GOTO redo;
       OUTSYMBOL(c);
END;
```

Or in ISO flavour[11]:

```
redo:
WHILE (INCHARACTER(0,c)) DO BEGIN
       IF c=65 THEN GOTO redo;
```

```
        OUTCHARACTER(1,c);
END;
```

In this case, the reading from an external source is an independent procedure, that always gives a result suitable for the cycle. Using `GOTO redo` (or `CONTINUE`) in different situations, that is directing backwards the program flow, causes repetition of code but this can generate closed loops (in Rutishauser's words), that is loops that end only by intervention by the user (by the CTRL-C keys combination).

11.8. LAYOUT OF DECLARATIONS WITHIN BLOCKS

Declarations must always be made at the head of a block, before any assignments, procedure calls, etc., in any order; the order respected by DEC system-10/20 ALGOL was:

1) scalars and arrays
2) procedures and switches (see Chapter 12).

but taxi removed this restriction. They are here written in case a program must be written for taxi *and* for the DEC system-10/20 ALGOL.

11.9. LAYOUT OF PROCEDURE CALLS WITHIN BLOCKS

Procedure calls that occur in a block should follow the declarations of variables that occur in the block. Consider the following example:

```
PROCEDURE P(X); VALUE X; REAL X;
BEGIN INTEGER J;
      ....
      J : = I;          ....
END;
...
Z:=P(3);
INTEGER I;
```

The assignment of `I` to `J` within the body of `P` utilizes the `I` that is declared following the call of `P`. The interpreter cannot "see" this `I` and, therefore, cannot take any rational action. In a case such as this, the user must declare `I` before the call of `P`:

```
PROCEDURE P(X); VALUE X; REAL X;
BEGIN INTEGER J;
      ....
      J : = I;          ....
END;

INTEGER I;
Z:=P(3);
```

If the user neglects to declare `I` before the call of `P`, the interpreter can easily detect the condition, because either `I` is unknown at the time of the assignment to `J`, whereupon an error message will occur when the declaration of `I` in the body of `P` is met, or there is a more global `I` available and, in this case, this last is used (and maybe changed, with unpredictable results).

[11] The previous pieces of code use the `INCHARACTER` and `OUTCHARACTER` procedures that are not ISO but ISO-like; the ISO ones are less friendly.

11.10. FORWARD REFERENCES

Although the DEC system-10/20 ALGOL interpreter operated in one pass and consequently, required the usage of the procedure FORWARD and is references, taxi operates in a completely different way, and FORWARD is no longer necessary..

A forward reference for a procedure had to be given when a procedure was called (either directly, or indirectly, through passing the procedure name as an actual parameter in a procedure call) before its body was encountered by the interpreter, for instance as in the case

```
FORWARD REAL PROCEDURE Q;

PROCEDURE P(X); VALUE X; REAL X;
BEGIN REAL Y;
       Y:= Q(X);
END P;

REAL PROCEDURE Q(Z); VALUE Z; REAL Z;
BEGIN REAL F;
       F := P(Z);
END Q;
```

Since Q was evaluated in P before its declaration, a FORWARD reference to Q had to be implemented. Reversing the two definitions would bring no help, since also Q calls P.

Since taxi operates a preliminary parsing of the source text to establish all references for WHILE and FOR cycles, for BEGIN-END blocks, for IF-THEN-ELSE structures and for PROCEDURE declarations, all procedures are stored in memory and are always available from the start, with the restriction for subsidiary procedures. (See Procedure Scopes 11.6.)

The conclusion is that, at present, the FORWARD statement and all characters until the terminator are simply ignored by taxi. The FORWARD procedure is maintained only for backward compatibility.

11.11. EXTERNAL PROCEDURES

The DEC system-10/20 ALGOL compiler could load during runtime already compiled procedures, attaching them to the program under compilation; this was achieved through the EXTERNAL procedure. The form of this keyword was the same as that of a FORWARD declaration, but with the word FORWARD replaced by EXTERNAL. For example:

```
EXTERNAL INTEGER PROCEDURE CALC
```

Such an EXTERNAL declaration could be made in any block within the program and had the same scope as if the procedure source appeared at that point.

Since taxi is not compiled, such a statement would be simply ignored and the line not loaded (there is no compiled procedure or program to be found); because of this, the EXTERNAL procedure in taxi was conceived with different scopes: if the EXTERNAL keyword is followed by a literal string.

E.g.

```
EXTERNAL "FILENAME";
```

taxi performs the following steps, during the loading of the program:

1. "FILENAME", which may contain a proper path, and which must be a textual ALGOL program file (usually with the .alg extension) is opened. The file name must be a literal string enclosed in double

quotes or double single quotes.

2. the file is copied in memory in the position of the EXTERNAL declaration (which is discarded) by expanding the memory. Nothing must precede or follow the EXTERNAL statement, which must reside alone in its program line.

These steps are executed before the preliminary process so that the external file is successively pre-parsed along with the rest of the program.

If "FILENAME" is not found, an error is raised and execution stops.

The external file is required to be a normal textual file; it should contain only PROCEDURE declarations and their bodies; in practice, it is a simple collection (library) of well-tested procedures. If some extra procedure blocks are found, they are loaded as well and later possibly executed, with unpredictable results.

It's responsibility of the programmer to call the loaded procedures with the correct number of arguments in the right order. It's far too easy to forget an argument or invert types without the declaration under your eyes!

For instance, a library with one factorial function could be stored in factlib.alg:

```
COMMENT FACTORIAL INSECURE RECURSIVE FUNCTION;
INTEGER PROCEDURE FACTORIAL(N);
     VALUE N; INTEGER N;
     IF N=1 THEN FACTORIAL := 1
     ELSE FACTORIAL:= N * FACTORIAL(N-1);
```

A practical example would be then, for instance in file fact.alg:

```
BEGIN
     EXTERNAL "factlib.alg";
     PRINT(FACTORIAL(7));
END.
```

If you build a library of well-tested procedures, their usage can be effective and practical if you should write anywhere in the listing statements like:

```
.....
EXTERNAL "math.alg";
EXTERNAL "string.alg";
.....
```

This guarantees that, if library procedures are well-tested, any error found is likely to reside in the current file and not in the library. This reduces the amount of debugging width. For this reason, libraries should print no error messages but should only return certain flags to signal the correct/incorrect computation; these flags are to be evaluated in the caller for the proper, driven error message printing or error-resolution code.

Some notes:

- This particular version of EXTERNAL is not compatible with the DEC system-10/20 ALGOL and does not belong to the ISO 1538-1984 document either.

- If you use option `--list`, you will see that the listed program will include the external libraries, loaded starting from the position of `EXTERNAL`, which is removed.
- For error-managing purposes, external libraries data are stored in a special memory, made available only in case of errors that require the printing of an error message. You can load at maximum up to 64 library files through the procedure `EXTERNAL`. This value is contained in the variable `EXTERN-LIBMAX` in `custom.h`; if you should need more libraries to load, change this value and recompile.
- The `EXTERNAL` procedure may be invoked as `INCLUDE`, with identical effects.

11.12. ADDITIONAL METHODS OF COMMENTARY

Three further ways of writing commentary are available to the user in addition to `COMMENT` and `!` described in Section 2.4.

11.12.1. Comment After `END`

Following the delimiter word `END`, the user may add any commentary, terminated by a semicolon, or the dot in case of the outer block's `END`; the characters that may appear in such a comment may be any, in any combination (even reserved words if necessary), except for the dot, the terminator words (see Table 2-4), and the comment words `!` and `COMMENT`. For example:

```
END OF PROC INVERT;
```

I repeat here that the terminator words `END`, `ELSE`, the dot `.`, the semicolon `;` and the comment words `!` and `COMMENT` cannot appear in the `END` commentary: their presence would put the interpreter in confusion (for block and `IF` structures and for comment inside a comment). So avoid using such characters in such commentary. Use synonyms like 'closure' for `END`, 'otherwise' for `ELSE` or 'remark' for `COMMENT` and use the comma and the colon or the question mark as punctuation signs. Use the dash − as the terminator (dot or semicolon).

11.12.2. Comments Within Procedure Headings

The ISO 1538-1984 specifies that any procedure heading may be enriched with specifiers (from the second argument on), like:

```
SPUR(A) ORDER:(N) RESULT TO:(S);
```

From a closing `)` to the following opening `:(`, any combination of characters (but the semicolon) may be used. The colon is required (also by the DEC system-10/20 ALGOL and the ISO documents) and must be followed by the open parenthesis of the following argument.

Even the call to such procedures may be as such enriched:

```
SPUR(VECTOR) ORDER:(7) RESULT IS:(RES);
```

This call is equivalent to

```
SPUR(VECTOR, 7, RES);
```

Please note that the final closing parenthesis is one (count the couples: they match) and that no space is allowed between one colon and the open parenthesis.

11.12.3. Comments after the last `END`

Any free form text, after the dot closing the outer block ended with last `END` onward, is ignored by the interpreter, so that any free-form text can be put here, program inputs and outputs, info, manual, help and so forth, also using the 'forbidden' characters `END`, `ELSE`, the dot and the semicolon, i.e.:

```
BEGIN      !outer block;
     ...
     ...
     ...
END OF OUTER BLOCK.<-- here begins the further free-form commentary

This program was created by John Smith on Oct. 23, 2016.
It can be used with the following data:
...
...
```

This free form text is NOT loaded in memory and can be used to store results or any kind of text being physically not part of the listing.

This feature can be used to isolate a final part of the program by using an intermediate `END.` (with the dot) put just before the part to be isolated.

12. SWITCHES

12.1. GENERAL

Switches enable the user to jump to one of the stored labels, depending on the value of an arithmetic expression, and in addition, provide automatic detection when such an expression is out of range for the switch.

12.2. SWITCH DECLARATIONS

A switch declaration takes the form of the word `SWITCH` followed by the name of the switch, an assignment (`:=`), and a list of labels, all on the same line (the declaration cannot be broken). These are called switch elements and must be in the scope of the switch declaration. For example:

```
SWITCH SW := LAB,L1,L2,OK,STOP;
```

A switch name must follow the usual rules of scope and, therefore, must not conflict with any local variable of the same name.

In addition to the example above, a switch element can also be one of the labels in another switch declaration.

E.g.:

```
SWITCH SW := LAB,L1,QUIT;
SWITCH TW := PROF,SW[I];
```

Here `SW[I]` and `TW[2]` are in practice the same label, through the value of `I` ranging from 1 to 3. Label `TW[1]` is instead a different and independent label.

Being a switch a collection of labels (i.e. strings), a string can be assigned to a switch element. Look a the following example:

```
STRING S;
SWITCH SW:=LAB1,LAB2;
S:=SW[1];
WRITE(S);
```

The previous piece of program prints `LAB1`, which is the label stored in `SW` at position one. This is because, in taxi, labels are stored internally in `SW` as strings, and an assignment to a string item is made then possible[12].

12.3. USE OF SWITCHES

A jump to a particular label in a switch declaration is made by following the word `GOTO` with the name of the switch and an arithmetic expression in square brackets. Thus:

```
GOTO SW[I];
```

where the variable `I` takes values from 1 to the `N`th label in the `SWITCH` list. This causes control to pass to the `I`th label in the switch declaration, unless `I` is negative or zero, or is larger than the number of switches in the switch declaration. In either case, there is no transfer of control (an error is raised). If the expression in square brackets is not of integer form, it is evaluated and rounded as usual. Consider the following more complicated example:

```
SWITCH SW := LAB,L1,L2,OK,STOP;
```

[12] I don't know if this is adherent to the DEC or ISO protocol, but I consider this an enhancement, not a bad feature.

```
SWITCH TW := L3,SW[J],L4;
....
GOTO TW[I];
```

If `I` has the value 3, a jump to `L4` occurs. If `I` has the value 2 and `J` has the value 1, a jump to `LAB` occurs, via `SW`.

More sophisticated switch elements are described in Chapter 14.

13. STRINGS

13.1. GENERAL

taxi (as the DEC system-10/20 ALGOL) includes a major extension to the string features defined in the Revised Report. Users wishing to run their programs on other compilers/interpreter should check whether the compiler they will use offers similar facilities. Scalar, array or procedure variables may be of type `STRING`, and are declared by the reserved word `STRING`. The string memory size, length and contents of string variables are defined via the various assignment statements described below.

Typical string declarations might be:

```
STRING S,T; STRING ARRAY SA[1:10];
STRING PROCEDURE B(X); VALUE X; REAL X;
```

13.2. STRING EXPRESSIONS AND ASSIGNMENTS

String expressions are limited to a single variable or array element, a string procedure call or a string constant. (For a full description of string constants see Section 4.5.) The only string operators are the comparison operators and the assignment operator. All other operations are achieved via the string library procedures described in Section 13.7.

String expressions can be assigned only to string variables. For example:

```
S:=T;
SA[I]:=SA[3];
SA[2]:=B(Z);
T:="ANY ""OLD"" IRON";
```

13.3. BYTE STRINGS

The value associated with a string variable is a sequence of bytes (in some ways, byte strings can be thought of as arrays of characters), up to 255 characters in length. Thus, when a byte string is created using a `READ` statement, the programmer need not know how long the string will be. The routine starts accepting characters after encountering the open quotation marks and continues until the close quotation marks have been read (or until the total length of 255 is reached).

When one string is assigned to another, as in the following example:

```
S:=T;
```

then a copy of `T` is made to which `S` will refer. Subsequent changes to the value of `T` will not affect `S`, or vice versa, unless a further assignment is made from one to the other.

13.4. BYTE SUBSCRIPTION

Byte strings can be modified through the byte subscription mechanism. Individual bytes in a string are referenced by following the string variable name by a decimal point and then the subscript number enclosed in square brackets. For example

```
S.[I]
```

refers to the `I`th byte of string `S`. The subscript may be any arithmetic expression and is evaluated in the same way as an array subscript of dimension `[1:N]`, where `N` is the length of the string.

Byte-subscripted string variables are regarded as being of type integer, having an integer value equivalent to the byte to which they refer. Therefore, to change the value of a particular byte in a string, a byte-subscript

must appear on the left-hand side of an arithmetic statement with the appropriate new value on the right-hand side.

If the new value is too large to be held in the byte, this is simply truncated. No warning is given.

If on the other hand, the subscript is out of the string memory space (less than 1 or greater than the maximum size) an error message is shown and execution stops.

13.5. NULL STRINGS

When a string is declared, a space of memory is assigned to it, filled with null values. Until a value is assigned to a string by the program, the string takes the null value, and any attempt to reference the string by a byte-subscript in the right-hand side of an arithmetic statement and within the string limits will yield zero.

Assignment with a byte-subscript on the left-hand side (and within the string limits) will store in the string memory that character value, but if any nulls should exist before it, this will be at all effects invisible.

E.g.

```
   S:="Art";
```

String picture:

```
  1 2 3 4 5 6 7 8 9...
 |A|r|t|0|0|0|0|0|0|
```

```
   S.[5]:=101; ! = 'e';  String picture:
  1 2 3 4 5 6 7 8 9...
 |A|r|t|0|e|0|0|0|0|
           ^
 inserted character
```

The insertion of character `'e'` in the middle of the string and beyond the null terminator is ignored; string `S` is anyway printed as `"Art"`.

But if the null is removed:

```
  S.[4]:=105; ! = 'i';
```

String picture:

```
  1 2 3 4 5 6 7 8 9...
 |A|r|t|i|e|0|0|0| |
         ^ inserted character
```

the character `'e'` becomes visible and string `S` is printed as `"Artie"`.

13.6. STRING COMPARISONS

Two byte-strings can be compared with each other using the usual comparison operators. For example

```
   IF S < T THEN GO TO L;
```

where `S` and `T` are string variables, string constants or string procedures. The effect of the comparison is to compare the strings byte-by-byte, the "lesser" string being that with the first lower value byte, working from left to right. Thus `"ABCD"` is less than `"ABCE"` or `"ABCDE"`. Where the strings to be compared are of different byte sizes, then the smaller bytes are regarded as being extended on the right by null bits.

Trailing blanks (spaces or tabs, or any mixture thereof) are treated as equal. Similarly, ASCII strings of different lengths will compare equally if the extra length comprises only spaces and tabs. In all other cases strings of unequal length can only be regarded as equal if the extra length consists entirely of blanks.

Thus, if `S` contains `"  ABCD  "` and `T` contains `"ABCD"` they are treated as equal, and a test such `S=T` would return true.

The cases of letters matter (upper or lower): they will compare equally if cases match. Thus if `S` contains `"A"` and `T` contains `"a"`, they are treated as different, and a test such `S=T` would return false.

13.7. STRING LIBRARY PROCEDURES

Sections 16.5 and 16.6 deal with the input and output procedures that apply to strings. What follows is the String Library Procedures implemented in taxi.

13.7.1. String concatenation

A string can be assigned the concatenated value of two (or more) strings with the procedure `CONCAT`. For example

```
S:=NEWSTRING(36);
S:=CONCAT(T,U);
S:=CONCAT(S,T);
```

If the sum of the lengths of the two strings were greater than the memory space of the result string, the bytes copied would be truncated as appropriate.

Thus, supposing `S` is a string of width 36, and that `S="123456789012345678901234567890"` with length equal to 30, and another string `T="1234567890"` with length equal to 10, the statement

```
S:=CONCAT(S,T);
```

would return in `S` the string `"123456789012345678901234567890123456"`, with the total length equal to 36 (its width) and the last characters of `T` silently cut out.

The original DEC system-10/20 ALGOL procedure admitted two strings only; taxi lets the user add a variable chain of strings, all separated by a comma:

```
S:=CONCAT(S,T," HELP ",S);
```

would return the string `S` plus the string `T`, plus the literal string addition `" HELP "` (spaces included) and another instance of `S`; provided the sum of the length of arguments is less than or equal to the length of the hosting string size, any sum is accomplished as desired.

This more-than-two-strings ability is an enhancement that the original DEC system-10/20 ALGOL and the ISO documents don't even quote.

If one string only is given to `CONCAT`, the result is this string. No error message is output:

```
S:=CONCAT(T);
```

is equivalent to

`S:=T` This feature is a natural consequence of the previous enhancement, extending the feature to the single case. I don't know what was the behaviour of the original DEC ALGOL.

13.7.2. String length and size

The primary attribute of a string, that is its length in characters, is returned by the integer procedure LENGTH.

Thus

```
I:=LENGTH(S);
```

would return the number of bytes of string S in I (whichever, depending on the context).

The procedure SIZE, instead, has a different usage than that in the DEC system-10/20 ALGOL. The bit size of a string, in taxi, is fixed to 8 bits (one byte). So the SIZE procedure has been redesigned to return the memory size of a string (the maximum width), that is the longest string it can host.

Thus

```
J:=SIZE(S);
```

would return 255 for a default string, or the value the string has after resizing it with NEWSTRING. The difference with LENGTH is subtle: while LENGTH tells us how many characters the string has now, SIZE tells us how many characters it can have at most. This is useful to check the memory space of a string and resize it before usage if needed.

13.7.3. String copy

A portion of an existing string can be assigned to another one using the string procedure COPY. This procedure can have one, two or three parameters.

Suppose the string T is assigned the value T:="THE VILLAIN AND THE PAINTER"; now the following cases can be considered:

1. The effect of COPY with one parameter is precisely the same as a simple string assignment, and this feature has been retained for the sake of continuity.

   S:=COPY(T); is equivalent to S:=T;

   Thus, for what about the previous assignment, S is assigned with "THE VILLAIN AND THE PAINTER".

2. Where there are two parameters, as in the following example:

   ```
   S:=COPY(T,M) ;
   ```

   where M is an arithmetic expression, then S is assigned the value of the first through Mth byte of T. Thus, if M yields 11, S is assigned with "THE VILLAIN".

3. If there are three parameters, as in the following example:

   ```
   S:=COPY(T,M,N) ;
   ```

   where both M and N are arithmetic expressions, then S is assigned the value of the Mth through Nth byte of T. Thus, if M yields 17 and N yields 27, S is assigned with "THE PAINTER".

   If values are invalid or beyond the scope (for instance M<0 or M>N), the assigned result is the empty

string. No warning is given.

As the examples show, the string count starts from 1.

13.7.4. String resize

A string can be resized by using this string procedure. NEWSTRING takes two parameters, the first being the number of null bytes to be assigned to the string, and the optional second (ignored by taxi but left for the sake of compatibility[13]) the characters size, and assigns the string new definition to the string on the left of the equal sign.

For example

```
S:=NEWSTRING(100[,7]) ;
```

causes S to be defined as a string of 100 ASCII characters; the previous memory space is destroyed and given back to the operating system. (The second argument 7, if given, is ignored.) The content of the previous memory is copied back (not beyond new string limit) to the new space. String S is 100 bytes long at this point (and thus byte subscripts up to and including S.[100] are valid).

13.7.5. String erasure

In Section 13.5 it was explained that if a null string is assigned to another string, then that string also becomes null, and the value previously held is lost. The procedure DELETE has the same effect on the string passed to it as a parameter, as the assignment of a null string would have, but while S:=T (with T being an empty string) would simply set S to the empty string, DELETE(S) would also erase all characters in the memory space of S.

NOTE: the length reference of the variable is not reset to 255 characters (if for instance it was resized with NEWSTRING). Thus, the statement

```
DELETE(S);
```

has the effect of erasing the string memory space of S maintaining its current structure.

If S is a string array, all instances of its elements (whatever dimension they have) are set to null. LENGTH would return 0 and SIZE would return the original memory size.

13.7.6. String extraction

Four string procedures for coordinated extraction are provided; these are HEAD, TAIL, TAKE and DROP. They are similar - but not identical - to the ones of the Unisys Algol. (See vol. 1, edition 2001.)

The procedure HEAD returns all the characters in a given base string, starting from the first character, that don't match any of the characters in a given match string.

E.g.:

```
S:="ABCDE";
WRITE(HEAD(S,"CE"));
```

would print AB because the character C of the match string stops the extraction; in the following case:

[13] taxi uses ASCII characters, which always are 8-bit numbers in the range 0÷255. The DEC system-10/20 ALGOL computer could treat characters of different sizes, and this is the reason for the second argument. (For instance, characters in the range 0÷127 use only numbers, lower and upper alphabetic characters and classic punctuation characters, and if this was the only need of the programmer, he/she could redefine a string which occupied less memory than the 8-bit size characters.)

```
WRITE(HEAD(S,"JE"));
```

would extract ABCD, because character J is not present in S, and the first break happens on character E.

The procedure TAIL is the complementary version of HEAD because it returns all the characters in a given base string, discarding all leading characters until a match in the given match string occurs.

E.g.

```
S:="ABCDE";
WRITE(TAIL(S,"CE"));
```

would print CDE because the character C of the match string stops the discarding and starts the extraction; in the following case:

```
WRITE(TAIL(S,"JE"));
```

would extract E, because character J is not present in S, and the discard proceeds until character E is met (and copied).

For the procedures HEAD and TAIL, the following scheme holds:

```
S is equivalent to HEAD(S,T) + TAIL(S,T)
```

The procedure TAKE returns a fixed number of characters in a given base string, starting from the first character; basing on the given length argument, as for example in the following lines:

```
S:="ABCDE";
WRITE(TAKE(S,2));
```

would print AB, that is the first two characters in S. If the index is lower than zero, or greater than the string length, it is resized accordingly.

The procedure DROP is the complementary version of TAKE, because it returns all the characters in a given base string, discarding the number of leading characters according to the length argument.

E.g.

```
S:="ABCDE";
WRITE(DROP(S,2));
```

would print CDE, because it would skip the first 2 characters.

For the procedures TAKE and DROP, the following scheme holds:

```
S is equivalent to TAKE(S,N) + DROP(S,N)
```

A final consideration: the Unisys procedure TAKE(S,N) is equivalent to the DEC system-10/20 ALGOL procedure COPY(S,N), with two arguments. Besides, the Unisys procedure DROP(S,N) is equivalent to the DEC system-10/20 ALGOL procedure COPY(S,N+1,M), where M is the length of string S (or greater value).

13.7.7. String from number

A number can be converted to its string representation by using the string procedure CONVERT. This procedure has one argument, which is the number to be converted, as a costant, a variable, an array item or a math expression:

```
S := CONVERT(N)
```

The number is converted to its proper form, that is an integer if it has no decimal part otherwise it is a decimal number; in case the REAL/LONG REAL is specified with a exponentation, the number is classified accordingly; the string representation is followed by the space terminator, and is preceded by the minus sign in case its negative, or by a space if positive. E.g., the following piece of code:

```
S := CONVERT(10.67&56);
OUTSTRING(1,S);
```

returns as expected:

```
1.067000000&+57
```

14. CONDITIONAL EXPRESSIONS AND STATEMENTS

14.1. GENERAL

taxi (as the DEC system-10/20 and ISO ALGOL) allows great flexibility in the construction of expressions and conditions.

Consider, for example, a variable `I` which could be set equal to 0 or 1 according to the value of a Boolean variable `B`: this could be written as:

```
I := 0;
IF B THEN I := 1;
```

Also, consider the case where a user wants to perform some action, depending on the value of `B`:

```
IF B THEN X1 := Y; IF NOT B THEN X2 := Y;
```

14.2. CONDITIONAL OPERANDS

taxi allows the user to substitute a conditional operand for any operand in an expression by the use of a construction involving

```
IF ..... THEN ..... ELSE.
```

For instance, the first example above can be rewritten

```
I := IF B THEN 1 ELSE 0;
```

This is more compact and of great use in cases such as:

```
J := J + (IF K < 1 THEN 1-K ELSE K-1);
```

Note that the conditional operand must always be bracketed if included in an expression; brackets may be avoided only when it forms the complete expression by itself.

In general, a conditional operand may replace an operand in any arithmetic or Boolean expression. Also, a conditional operand may replace a label and act as an element in a switch list, for example:

```
SWITCH SW := L1, IF B THEN L2 ELSE L3, L4;
```

It is also permitted in an array subscript or in a byte subscript;

E.g.

```
X := A[I, IF L = 0 THEN J ELSE J+1];
```

It must be underlined here that a conditional expression is made of the `THEN` *and* the `ELSE` part, i.e. the `ELSE` part is not optional, because a value must be ensured, either as a result of the `THEN` or as a result of the `ELSE` part.

Besides, the expression `IF..THEN..ELSE..` is interpreted a value, and must reside on the same line: it cannot be broken. An error is printed if `THEN` or `ELSE` are not found.

14.3. CONDITIONAL EXPRESSIONS AND STATEMENTS

Since a conditional operand may replace any operand in an expression, operands can also be replaced in conditional expressions. Consider the following example:

```
IF (IF B THEN B1 ELSE B2) THEN I := I + 1;
```

The IF–THEN–ELSE between brackets may be seen as a number. It is the same sequence as if it was written as

```
C:=B2;
IF B THEN C:=B1;
IF C THEN I := I + 1;
```

where C takes the value B1 or B2 depending on the value of B.

14.4. CONDITIONAL STATEMENTS

The reader was introduced to conditional statements of the form

```
IF B THEN S1 ELSE S2
```

in Chapter 7. The full power of this type of statement can now be demonstrated. First, S1 and S2 can be compound statements or blocks. For example:

```
IF I < 0 THEN
   BEGIN
      I := -I; B:= FALSE
   END
ELSE
   BEGIN
      I := I + 1; GOTO L2
   END;
```

Second, the whole structure of the IF ..... THEN ..... ELSE statement can be made more powerful by using conditional statements within themselves. For example:

```
IF X < 0 THEN X := 0 ELSE IF B THEN GOTO L
```

This is equivalent to the following sequence of statements:

```
     IF NOT X < 0 THEN GOTO L1;
     X := 0; GOTO L2;
L1:  IF NOT B THEN GOTO L2;
     GOTO L;
L2:
```

The former method of expression is both briefer and more elegant. Conditional statements take the general form

```
IF B THEN S1 ELSE S2
```

where S1 and S2 may both be conditional statements. However, if there is any ambiguity, bracketing using BEGIN and END must be used to clarify this. Consider the following example:

```
IF B THEN IF X = 0 THEN Y := Z ELSE P := Q;
```

This piece of code can be seen as

```
IF B THEN
   BEGIN
      IF X = 0 THEN Y := Z
   END
ELSE P := Q
```

or

```
IF B THEN
   BEGIN
     IF X = 0 THEN Y := Z ELSE P := Q
   END
```

The first case is interpreted as:

```
     IF NOT B THEN GO TO L1;
     IF NOT X = 0 THEN GO TO L2;
     Y := X: GO TO L2;
L1: P : = Q:
L2:
```

The second case is interpreted as:

```
     IF NOT B THEN GOTO L2;
     IF NOT X = 0 THEN GOTO L1:
     Y := Z; GO TO L2:
L1: P := Q:
L2:
```

ALGOL 60 forbids such ambiguities by forbidding the sequence

```
THEN IF .... THEN .... ELSE;
```

taxi, in particular, stops execution if such a condition appears in the listing; if you need an `IF` after `THEN`, enclose the `IF` into a `BEGIN-END` block:

```
IF .... THEN
     BEGIN
       IF ....
     END;
...
```

14.5. DESIGNATIONAL EXPRESSIONS

A designational expression is something that acts as an argument in a `GOTO` statement, either directly, or indirectly via a formal procedure parameter of type label. This may simply be a label or a switch element. Thus the following are designational expressions:

```
L
IF B THEN L1 ELSE L2
```

```
IF X < 0 THEN SW[I] ELSE IF X+Y >= Z THEN TW[J] ELSE L
```

These designational expressions would be used in the following manner:

```
GO TO L;
GOTO IF B THEN L1 ELSE L2;
GOTO IF X < 0 THEN SW[I] ELSE IF X+Y >= Z THEN TW[J] ELSE L;
```

15. OWN VARIABLES AND ARRAYS

15.1. GENERALITIES ABOUT OWN VARIABLES

OWN variables are a special kind of ALGOL variables and may be of type integer, real, long real, Boolean or string, either scalar or array.

The variables have the following properties:

1. Although following the normal scope rules, the variables are not recursive; they are the same copy of each variable that is being used in all occurrences of a procedure or block.

2. When control passes out of a block, the values are retained and are still available when the block is re-entered.

3. The initial value is set to zero before the execution of the program. (FALSE in the case of Boolean OWN variables). OWN strings are initialized to the empty string.

OWN variables are declared by writing the usual declaration with the word OWN preceding it. For example:

```
OWN INTEGER I,J,K;
OWN REAL ARRAY THETA[l:M];
```

15.2. OWN ARRAYS

OWN arrays are implemented in a completely dynamic fashion in taxi ALGOL. The declaration proceeds according to the following rules.

1. If this is the first time the array is declared, space is obtained and then the array laid out. If the array has been laid out before, proceed to Step 2.

2. The bounds are examined to ensure that these are identical to the ones of the previous construction of this array and the array is left unaltered if found to be of the same dimension; otherwise, proceed to Step 3.

3. A new array is constructed and the common elements if any, are copied from the old array; the remaining elements are zeroed. The old array is then deleted and the allocated space is recovered for future use.

For example, if an OWN array A is declared as follows:

```
OWN REAL ARRAY A[l:M,M:N];
```

where M is 2 and N is 5 the first time, and M is 1 and N is 4 the second time, the elements [1,2], [1,3] and [1,4] are copied over, and the remaining elements of the new array are zeroed. The memory is copied as a whole, and the new array can have elements in different positions.

NOTES:

- in the DEC system-10/20 ALGOL manual or the ISO 1538-1984, there is no mention of implicit real type in case of OWN variables if the type is not specified. But taxi, as for the arrays, in case OWN is used alone, defaults to OWN REAL;

- please note that declaring `OWN` variables in the main block (the most external block ending with `END` plus dot) is useless, and is equivalent to a declaration without the `OWN` keyword.

16. DATA TRANSMISSION

16.1. GENERAL

Data transmission encompasses the input and output of data between the user's program and the peripheral devices. (Originally they included disks, tapes with DEC formatting, magnetic tapes, card readers, card punches and line printers.)

taxi provides the user with a default channel for input (channel zero) and a default channel for output (channel 1). These cannot be changed and are always active.

taxi also provides the user with a set of pre-installed basic procedures for uniformly handling data. All available peripheral devices (at present DSK, TTY and PRN/LPT/PLT) and all file channels that taxi can handle are under the user's control completely and can be allocated or released at any time throughout the execution of the program. The user can handle up to physical 14 devices simultaneously, numbered 2 to 15 (16 including devices 0 and 1).

16.2. ALLOCATION OF PERIPHERAL DEVICES

Peripheral devices are allocated to the user's program by calls to the library procedures INPUT or OUTPUT. A call to one of these procedures usually has two parameters. The first is the channel number, an integer in the range 2 to 15, on which the device is to operate. Only one device at a time may be operated on a channel since a channel provides either input *or* output facilities. The second parameter is either a string or a string constant. The text contained in the string is the logical name of the device to be allocated to this channel.

NOTE: at present, only TTY and DSK can be used for input, and TTY as the monitor, DSK, PRN/LPT/PLT as a deferred printing can be used for output. The following Table 16-1 lists all devices:

Table 16-1
Standard Device Names

Device Name	Peripheral
DSK	Any disk mounted on the File System

NOTE: This device should be used even for external HD or USB keys, or any other mountable directory-driven device, since they are simple directories for a UNIX File System.

Device Name	Peripheral
PRN - LPT - PLT	Line printer/Plotter

NOTE: These devices identify current printer/plotter attached to the machine. taxi is unable at present to select a specific printer. Printing is deferred until the end of the program or a specific channel RELEASE, and printed afterwards; in the meantime data are stored into a temporary file.

Device Name	Peripheral
TTY	The terminal

NOTE: This device means input governed by keyboard and output directed to the monitor. Input from channel 0 and output to channel 1 are both protected; on channels 2÷15, instead, input OR output can be done.

For example, to allocate the terminal for use as an input device on channel 5, the user would use the statement

```
INPUT(5, "TTY");
```

or, if R were a string possessing a byte string that has the characters TTY (or tty) in it,

```
INPUT(5, R);
```

NOTE: All devices are allocated to operate in one direction only; thus, if the user wants simultaneously to input and output from a device (included the terminal), two separate channels must be used.

NOTE: the INPUT and OUTPUT logical feature is provided. See the paragraph "LOGICAL INPUT/OUTPUT."

16.2.1. Device Modes

NOTE: the third parameter (device modes) has no meaning here and, if used, it is simply ignored.

16.2.2. Buffering

NOTE: the fourth parameter (buffering type) has no meaning here and, if used, it is simply ignored. Buffering is left to the underlying UNIX Operating system.

16.2.3. Error Returns

Normally, if the device allocation fails (for example if the device is in use by another job), a suitable message is typed and the program terminated. The user can prevent this by providing, as the fifth parameter to INPUT or OUTPUT, a label to which control is to be passed in the event of an error. For example:

```
OUTPUT(14,"DSK",O,O,ERROR.LABEL);
```

If the actual label parameter is a switch whose subscript is out of range, or it is an inexistent label, the procedures behave as though the label parameter were absent.

NOTE: The third and fourth parameters must be specified in this case. They are ignored anyway.

16.3. FILE DEVICES

Being everything a file, all peripheral devices, such as disks or tapes, after being chosen as INPUT/OUTPUT with the DSK identifier, require the opening of a specifically named file before any input or output operations can be performed. The opening of this file is performed using the procedure OPENFILE, which is called after the device has been allocated to a channel. The procedure call has two main parameters:

- the channel number on which the device has been allocated;

- a string variable possessing a byte string or a string constant, the text of which is the name of the file.

The DSK device can be used for simultaneous input/output channels on several files on the File System.

The user can also specify a permission code and/or a flag for erasing the file once opened. For example, to open a file named "TEST.DAT" on channel 9 with permissions 744 and anew (owner read/write/execute and group/world read-only), the user could write

```
OPENFILE (9,"TEST.DAT",%744,N);
```

The octal code specified as the permission code works according to the following table (UNIX permissions):

```
4000    set user id on execution
2000    set group id on execution
```

```
1000    save swapped text even after use

0400    read for owner
0200    write for owner
0100    execute for owner

0040    read for group
0020    write for group
0010    execute for group

0004    read for world
0002    write for world
0001    execute for world

0000    user usage
```

For instance, to specify reading permission for all, and read/write permission for the user only, one could use `%644` as the permission code; of course, any integer can be used in this place, but the octal form (preceded by `%`) makes designing the permission code *as-is*. If a null value is passed, it is left to the shell umask code the task to define how files are created in the user directories.

The fourth argument is any number, whose integer part is taken: if null, the file is not erased, otherwise it is. For instance

```
OPENFILE(9,"TEST.DAT",0,1);
```

creates file `TEST.DAT` in the current directory, with null permission codes, and erased before it is used.

NOTE: the file must exist when the `OPENFILE` procedure is called.

When operations on a file are over, the file should be closed. A file is closed by using the procedure `CLOSEFILE`, with a list of parameters, each being a channel number on which the file is open. Thus,

```
CLOSEFILE(9);
```

closes the file that is open on channel 9, whereas

```
CLOSEFILE (3,9);
```

closes the files that are open on channel 3 and 9.

NOTE: when the program ends, taxi as a safety measure, takes care of closing all files left opened. But it's, anyway, a good programming style closing a file after use.

The user can also rename or delete existing files: if a file is already open, use of `OPENFILE` causes the file to be renamed with the new name supplied. Thus the sequence

```
OPENFILE (5,"TESTl.DAT");
OPENFILE (5,"TEST2.DAT");
```

causes the file with name `TESTl.DAT` to be renamed `TEST2.DAT`. If the string containing the new name is null, the original file is erased. Thus,

```
OPENFILE (5,"TEST3.DAT");
```

```
OPENFILE (5,"");
```

causes the file TEST3.DAT to be erased. (The empty file remains in the file system, and should be deleted manually.)

16.3.1. Error Returns

Normally, if the requested operations fail (for example an input file is not a regular file or it does not exist), a suitable message is typed and the program terminates. The user can prevent this by providing a label and an optional integer variable name as the fifth and sixth parameters to OPENFILE. In the event of an error, control will be passed to the label, with an error-code set into the integer variable if present.

```
INPUT(5,"DSK");
INTEGER N;
OPENFILE(5,"TEST.DAT",0,0,LAB1,N);
....
LAB1: COMMENT ERROR-TREATMENT SECTION;
IF N=44 THEN ....
```

NOTES:

- The error-codes are those returned by option `--errors-list`.
- The third and fourth parameters must be present if the error return parameter is specified. Defaults will be taken if both parameters are specified as zero.
- If the actual label parameter is a switch whose subscript is out of range, an error message is printed and the program stops.

The error code variable can be used to check what kind of error occurred and take the appropriate action to avoid it repeats. The ISO 1538-1984 procedure FAULT is useful with this regard to print the associated error message. Make sure that the variable used in OPENFILE as the sixth parameter refers to a label currently in scope.

16.4. CREATING A NEW EMPTY FILE

The procedure CREATEFILE (not belonging to the original DEC system-10/20 ALGOL) was designed to enable the creation of a fresh new empty file on the file system. Since procedure OPENFILE acts only on existing files, provided they are previously created in the underlying Operating System or created empty during runtime before usage (for instance, before a TRANSFER). You can create the file by yourself, before executing the program or use CREATEFILE in the source.

The file name must be given along with the whole path. Example:

```
CREATEFILE("/path/to/file/name.XX");
```

The procedure CREATEFILE is not associated with a channel. It acts on the DSK device, depending on current working directory and pathname. The file is created and closed, so no action is performed upon it. To use it as an output file, OPENFILE and SELECTOUTPUT (see next chapter) must be used to connect the newly created file to the program and select it as the output source.

NOTE: If the file should already exist, it is erased and retained empty. So watch out.

16.5. SELECTING INPUT/OUTPUT CHANNELS

Before a user uses a device to transfer data, assuming that the device has already been allocated to some channel (or the file has already been opened), the appropriate input or output channel must be "selected" for use as the input or output channel. All data input and output always occurs on the currently selected input

channel and output channel, respectively. The user may change the selection of channels at any time, switching from one channel to another without loss of data, irrespective of whether complete lines (or records) of data have been read or not. taxi does not assume any structure in the data: all input and output channels are regarded as pipelines through which the user pulls or pushes data.

To select an input channel, a call to the procedure `SELECTINPUT` must be made. This has one parameter, which is the channel number. Thus

```
SELECTINPUT(5);
```

causes input channel 5 to be selected.

Similarly, the procedure `SELECTOUTPUT` is used to select an output channel.

```
SELECTOUTPUT(6);
```

NOTE: `SELECTOUTPUT` opens the file always in append mode (that is, the writing begins at the end of the existing data). To ensure rewriting all, use the erase feature of `OPENFILE`. (See FILE DEVICES.)

16.6. RELEASING DEVICES

The procedure `RELEASE`, with a list of parameters, each being a channel number, is used to release a device from a channel. Thus,

```
RELEASE(5);
```

releases the device or file allocated to channel 5, whereas

```
RELEASE(3,7);
```

releases the devices or files allocated to channels 3 and 7.

NOTE: the multiple argument feature was not on the DEC system-10/20 ALGOL.

NOTE: In case of files (`DSK` devices), `RELEASE` must be used only referencing to channels already closed with `CLOSEFILE` (or never opened with `OPENFILE`). Moreover, `RELEASE` won't act on channels 0 and 1, which cannot be released.

If a user terminates his program without releasing used channel devices, these are automatically released and all file closed by taxi when the program terminates.

The procedure `RELEASE` can also close the channel associated to a string (logical device); e.g the syntax:

```
RELEASE(26);
```

cause the string, that it was connected to as an input/output channel through `INPUT` or `OUTPUT`, to be detached from the channel; the string is not altered anyway.

16.7. BASIC INPUT/OUTPUT PROCEDURES

16.7.1. Byte Processing Procedures

The following the DEC system-10/20 ALGOL procedures may be used with any device to handle 8-bits bytes, normally used with devices supplying or accepting ASCII bytes (and thus called "symbols").

1. `INSYMBOL(S);` - (where `S` is usually some integer variable) causes the next byte to be read from the currently selected input channel and stored in `S`.

2. `OUTSYMBOL(J);` - (where `J` is usually some integer expression) causes the value of `J` to be output as a byte to the currently selected output channel. If `J` is too large for the byte size of the device in use, it is truncated to the byte size.

3. `NEXTSYMBOL(S);` - acts in the same way as `INSYMBOL()` except that the byte pointer for the input channel is not advanced to the next available byte. This gives the user a look-ahead facility of one byte.

4. `SKIPSYMBOL;` - causes the next byte from the selected input channel to be read and ignored.

5. `BREAKOUTPUT;` - causes all bytes in the buffer of a file output device to be sent immediately to it. This ensures the file buffer is emptied before eventually setting it for input.

See also chapter 18.8 about the two different philosophies behind the DEC system-10/20 ALGOL and ISO 1538-1984 documents.

16.7.2. Miscellaneous Symbol Procedures

The procedures `SPACE`, `TAB`, `PAGE`, and `NEWLINE` cause the appropriate number of spaces, tabs, page throws, or new lines to be output. This number is specified by a single integer parameter. If the parameter is omitted a value of one is assumed. Thus

```
SPACE(5);
```

causes five spaces to be output, whereas

```
SPACE; or

SPACE(1);
```

cause one space to be output.

NOTES: - `NEWLINE` is ignored if current `OUTPUT` is a logical channel. (See 16.8.)
- `PAGE` currently clears the screen of the monitor output. It is without effect on files.

16.7.3. Numeric and String Procedures

Numeric procedures are used to read and print numeric quantities. The procedures will normally be used with a device that is operating in ASCII mode, and are capable of processing integer, real, or long real quantities in fixed-point and floating-point representation.

16.7.3.1. Numeric and String Input

Numeric data for input can be represented in any format that would be acceptable as a numeric constant in a program, irrespective of the type of variable involved. When a number is read, automatic type conversion is performed, giving a result of the same type as if an assignment of the data represented as a constant in the program had been executed.

There is a minor restriction in that no spaces, tabs, or other non-printing symbols may appear in such numeric data. Otherwise, any symbol that is not a part of a numeric quantity may act as a terminator for such a quantity. It is strongly recommended that spaces, tabs, commas or new lines be used as separators.

For example:

```
3.4   -9.6   1.36   -52

0,     14.9
```

NOTE: In reading a numeric quantity, the terminating symbol, that is the first symbol that is not part of the number, is lost.

taxi also allows the user to input floating-point data written in FORTRAN format, that is using `E` for `&` or `@`, and `D` for `&&` or `@@`. However, since numbers are stored in fixed memory space, this is of little importance for what about the input phase and is only meaningful in output, where the type of the variable influences the number picture.

The procedure `READ` is used to input numeric data and also strings. This procedure may have any number of parameters (up to the end of a physical program line), of type integer, real, long real, Boolean, string or label.

The effect is as follows:

1. For integer, real and long real variables, a number is read and converted to the type appropriate to the parameter and then assigned to the variable.

2. For Boolean, a number is read as if for an integer variable and assigned to the variable.

3. For a string or label variable, the data text is scanned until a quote (`"`) or (`''`) is found, and the text following this up to but not including the next free quote is read in and a byte string is generated, which is then possessed by the string or label variable.

If the sequence `""` is found inside the string, a single `"` is stored, and reading of the string continues. Of course `""` indicates, per se, the empty string.

16.7.3.2. Numeric Output

Numeric data is output using the procedure `PRINT`. This procedure may have one, two, or three parameters, the first of which is the variable to be printed. This variable may be a Boolean, integer, real, or long real. A pure literal number (having or not the decimal point) is always interpreted as real. The second and third parameters determine the format to be used and are integer expressions. If omitted, both parameters are assumed to be zero. The effect of the various combinations of the format integers, `M` and `N`, is as follows:

```
    M>O, N>O:        Fixed-point printing, M places before the decimal
                     point, N places after. A sign, space if positive,
                     dash if negative appears before the number. Zeros
                     before the decimal point (unless the rightmost)
                     are replaced by spaces and the sign moved up to
                     the number. A space character appears after the
                     last digit.

                     This format always outputs M+N+3 symbols. Example:

PRINT(3,3,3);PRINT(3.2,3,3);PRINT(0.00003,3,3);

   3.000    3.200    0.000
```

```
   M>O, N=O:        The same as the preceding except that (1) no
                    fractional part appears, and (2) the decimal point
                    is suppressed.

                    This format always outputs M+l symbols. Example:

PRINT(3,3,0);PRINT(3.2,3,0);PRINT(0.00003,3,0);

   3    3    0
```

Please note the third number, basing on the preceding laws, is printed as zero.

```
   M=O, N>O:        Floating-point format, consisting of a sign, a
                    decimal digit, a decimal point, N more decimal
                    digits, an exponent consisting of & for real,
                    && for long real followed by the exponent sign and
                    an exponent, zero suppressed from the left with
                    1 to 3 digits and a final space after the last.
                    (Note: the exponent sign is always printed.)

                    This format outputs at most N+8 characters for
                    real and N+9 for long real quantities.

                    NOTE: the exponent character & in the number
                    picturing can be changed by means of option
                    --exp=C, where C is the chosen character between
                    E, & and @. Example:

PRINT(3,0,3);PRINT(3.2,0,3);PRINT(0.00003,0,3);

 3.000&+0  3.200&+0  3.000&-5
```

If only two parameters appear, format `M,0` is assumed for integer variables, and format `0,N` for real and long real quantities, where `M` and `N` take, respectively, the value of the second parameter. Example:

```
PRINT(3,3);PRINT(3.2,3);PRINT(0.00003,3);

  3.000&+0  3.200&+0  3.000&-5
```

If only one parameter appears, the format is interpreted as `0,0` which assumes standard printing modes of `11,0` for integer quantities, `0,9` for real quantities, and `0,17` for long real quantities. Example:

```
PRINT(3);PRINT(3.2);PRINT(0.00003);

 3.000000000&+0  3.200000000&+0  3.000000000&-5
```

If the user should request more digits to be printed than those significant in real or long real numbers, the appropriate number of zeros follow a properly-rounded print of the number to the maximum precision available, with possible garbage digits at the end of the decimal scale.

The `PRINTLN` procedure works exactly as the `PRINT` statement, but outputs a Carriage Return after the number. This procedure was not in the DEC system-10/20 ALGOL but comes in handy when you want to avoid all the `NEWLINE` statements needed to go to the next line.

16.7.3.3. String Output

A byte string may have its contents transferred to the currently selected output channel using the procedure `WRITE`, whose single parameter is either a string constant or a string variable containing the string to be output. For example:

```
WRITE(S);
```

or

```
WRITE ("THE MOON IS MADE OF GREEN CHEESE");
```

With exceptions explained in the following paragraphs, all of the bytes in the string are output literally, retaining the upper and lower format.

NOTE: Unlike some other ALGOL implementations, spaces and other non-printing symbols in byte strings are meaningful in taxi.

Special editing characters are permitted within square brackets within the text of a byte string. These have a special function:

`P`	`Page throw`
`C or N`	`Newline (C stands for carriage return, line feed)`
`T`	`Tab`
`S`	`Space`
`B`	`Break output`

Any combination of these characters, with optional preceding repetition counts, can appear within square brackets in a byte string and are output as their special interpretation demands. For example:

```
WRITE("ABCD[P2C5S]EFGH");
```

causes the following to be output:

1. the symbols `ABCD`
2. one-page throw, two new lines and five spaces
3. the symbols `EFGH`.

That is:

```
ABCD


     EFGH
```

To output the symbols

```
[ ] " or ;
```

these must appear in the form

```
[[ ]] "" or ;;
```

respectively. Thus

```
WRITE(''""A[[I]] := 3;;""'');
```

causes the text

```
"A[I] := 3;"
```

to be output. In the string, the double-single quote delimiting characters were used to help distinguished inner quotes from outer quotes. The inverse (and equivalent) is:

```
WRITE("''A[[I]] := 3;;''");
```

which outputs the following

```
''A[I] := 3;''
```

Please note that the single quote represents itself, and need not be doubled.

The WRITELN procedure works exactly as the WRITE statement but outputs a Carriage Return after the number. This procedure was not in the DEC system-10/20 ALGOL but comes in handy when you want to avoid all the NEWLINE statements or the [C] inner printing codes needed to go to the next line. This may not portable, though.

16.7.3.4. Octal Input/Output

The procedures READOCTAL and PRINTOCTAL, respectively, allow the user to input and output quantities written/typed in octal format.

On input, for single precision variables, up to 11 octal digits are read, preceded by the symbol %, the terminator being any non-numeric symbol in the range 0−7. For long real variables, two such octal numbers must be presented for input, the first preceded by the symbol % and the second optionally preceded by the symbol %.

On output, 11 octal digits, preceded by the symbol %, are printed for single precision variables. For long real variables, two quantities each with 11 octal digits globally preceded by the symbol % are printed, separated by a space. The output is followed by a terminator (the space character).

The PRINTOCTAL procedure has one variable parameter which may be of type integer, real, long real or Boolean. The READOCTAL procedure may have any number of variables, separated by a comma (as for READ, see).

For instance, the program

```
BEGIN
    LONG REAL A,B;
    A:=23455;
    PRINTOCTAL(A); NEWLINE;
    PRINTLN(A);
    WRITE("ENTER AN OCTAL NUMBER: ");
    READOCTAL(B);
```

```
    PRINTOCTAL(B); NEWLINE;
    PRINTLN(B);
END.
```

produces the following output

```
 %00000000000 00000055637
 2.345500000000000036&&+4
ENTER AN OCTAL NUMBER: %00000000000 00000055637
 %00000000000 00000055637
 2.345500000000000036&&+4
```

(This example supposes to re-enter, at the READOCTAL request, the very same number previously printed by PRINTOCTAL.)

16.7.3.5. Array input/output

Following what Heinz Rutishaser suggests in his 1970 introduction to Algol, I added to taxi the two array procedures INARRAY and OUTARRAY, according to the ISO style. These procedures work with numeric and string arrays.

```
INARRAY(ch,ar);
```

The previous procedure causes the array ar to be filled with values coming from the input channel ch. The array ar may be:

* a Boolean array, in which cases values are converted to zeros or ones.
* an integer array, in which cases values are cut to integers.
* a real array or a long real array, in which case the complete values are retained.

While evaluating numbers from the channel ch, INARRAY considers as a literal number any sequence of characters in the form:

```
SNNN[.NNN[SCNN]]
```

where S is the sign '−' or '+' (the latter being optional), N is any digit 0..9, the dot separates the integer part from the fractional part (the latter two being optional), C is the exponential character '&', '@' or 'E'; 'D' is also accepted because of the Fortran double-precision format. The number must begin with a sign, a digit, a dot or the '&' or '@' characters. ('E' and 'D' are not available as starters.) The exponential group SCNN must be given as a whole (the sign S being optional). Expressions are not evaluated.

Any sequence of alphabetic or punctuation characters that does not start a valid numeric sequence is read and discarded. In particular, stating on the previous rules, a sequence like E4 (which could be interpreted as 1*10^4=10000) is equivalent to 4, because the character E is read and discarded; analogously, the expression SQR(3) is equivalent to 3 because SQR(and) are read and discarded.

```
OUTARRAY(ch,ar[,width]);
```

The previous procedure causes the array ar to be printed on the screen to channel ch, regardless of the number type; in case the number cannot be represented in floating-point format, it is printed in exponential form (in this case, the figure SCNN is included in the printable width). The third optional parameter is an integer expression that by default is 12 characters, that includes the sign (1 character), the integer part (variable), the dot (1), the decimal part (variable), the exponential (4, where needed) and a final space (1). This number can be used to size the output precision and alignment, extending or reducing the precision to the

desired depth, with the following set:

`width<4`	only the integer part is printed
`width=5`	the integer part plus 1 decimal are printed
`width=6`	the integer part plus 2 decimals are printed
and so on...	

Please note that stating on the previous rules, long real arrays are not printed using their proper type-precision (as it would with `PRINT` or `OUTLONGREAL`), but using appropriately the `width` parameter.

Arrays are printed with one blank line preceding and one blank line following; the printing rules follow their dimensions:

* if the dimension is 1, the array is a vector and is printed vertically.
* if the dimension is 2, the array is a matrix and is printed in a rectangle.
* if the dimension is N > 2, the array is printed in a series of consecutive matrices. It's the user responsibility to separate the output.

16.8. SETTING DEFAULT INPUT/OUTPUT

If the user does not select any input or output channels, input and output occur via channel 0 from the keyboard (input) and 1 to the user's terminal (output). Thus, for simple programs where the user wishes to input a few numbers and print a few results, he simply uses `READ`, types in the data online through his terminal, and gets back the results from `PRINT`.

16.9. SETTING LOGICAL INPUT/OUTPUT

In addition to the 14+2 channels used to communicate with peripheral devices, san additional 16 channels, numbered from 16 to 31, are provided. These are input or output channels that use byte strings as a means of storage.

Using the procedures `INPUT` or `OUTPUT`, the user can attach a channel to a byte string possessed by a string variable, and can read and write bytes from and to this byte string, either to and from a peripheral device or to and from another byte string.

```
INPUT(20,S) ;
```

or

```
OUTPUT(20,S) ;
```

cause the byte string possessed by the string variable `S` to be used as logical channel 20; this channel may subsequently be selected for input or output, as appropriate.

The user is still free, of course, to manipulate the individual bytes within the byte string utilizing the byte-subscription facilities available. Such facilities enable the user to read a file from a peripheral device into a string, process it in any way whatsoever, and output it again.

NOTE: when the `INPUT` or `OUTPUT` process of reading-from/writing-to a string reaches the string limit space, the buffer pointer is reset, causing the string to be re-read or re-written. If you try to print a string longer than the remaining space (starting the count from the buffer pointer to the string limit), only the part that fits is copied, the buffer pointer is reset and the rest is copied with the same cautions. Take it in account when you work with logical devices, and assure you have sufficient space on the string (if in case use `NEW-STRING`. See 13.7.4).

16.10. I/O CHANNEL STATUS

The status of any input or output channel can be determined at any time using the Boolean procedure IOCHAN, which takes an integer channel number as the parameter. The status returned is bit coded as in Table 16-2:

Table 16-2
I/O Channel Bit Status

Bit	Value	Meaning if Set
18	%400000	Device is physical (i.e., not logical)
17	%200000	Directory device (unused)
16	%100000	Terminal device
15	%040000	ASCII mode (always set, unmodifiable)
14	%020000	Magnetic tape (unused)
13	%010000	Plotter/Printer
12	%004000	Set for default TTY on channel 0
11	%002000	Device is spooled (always set, unmodifiable)
10	%001000	Device can do input
9	%000400	Device is initialized for input
8	%000200	File is open for input
7	%000100	End of file encountered
6	%000040	Input status OK
5	%000020	Device can do output
4	%000010	Device is initialized for output
3	%000004	File is open for output
2	%000002	Device quota exceeded (unused)
1	%000001	Output status OK

Some of these bits are of little or no use to the programmer, but, for example, if a channel connection is established, and the programmer does not know whether or not the device is ready for input or output, IOCHAN can be used to determine this.

The following example shows how the user can handle an unknown device whose name is given to the program via the user's terminal, and print the first line of the textual file if it's available:

```
BEGIN
    STRING DEVICE, FILE, ALINE; INTEGER CHANNEL;

    WRITE ("CHANNEL NO: "); BREAK.OUTPUT;
    READ (CHANNEL);

    WRITE ("[C]DEVICE NAME: "); BREAK.OUTPUT;
    READ (DEVICE);
```

```
    INPUT (CHANNEL, DEVICE);
    IF IOCHAN (CHANNEL) AND %200000 THEN BEGIN
          WRITE (" [C]FILE NAME: "); BREAK.OUTPUT;
      READ (FILE);
           OPENFILE (CHANNEL, FILE);
          SELECTINPUT(CHANNEL);

      COMMENT READ & WRITE THE FIRST LINE OF FILE;
          READ(ALINE);  WRITELN(ALINE);
          CLOSEFILE(CHANNEL);

    END
    RELEASE(CHANNEL);
END.
```

NOTE: When using Boolean expressions involving `IOCHAN`, the rules for evaluation in this implementation should be kept in mind. See section 5.2.

NOTE: An end-of-file on input is, at present, considered only for the `DSK` device. The EOF flag is set when reading from a text file has reached and passed the last character. The `TTY` does not check for end-of-file on input.

NOTE: An end-of-file on output is at present ignored. taxi does not control the exceeding of a disk quota with files, or the exceeding of the disk memory, because this is left to the hosting Operating System. In this case, appearing error messages depend on the file system.

NOTE: In case of logical devices, there is no end-of-file, neither in input nor output; when the end of the string is reached, the string pointer is reset to the start of the string, and the reading or writing can go on, in a circular model. This is a minor deviation from the DEC system-10/20 ALGOL behaviour.

16.11. TRANSFERRING FILES

Once devices have been allocated to an input and an output channel, a complete file of information may be transferred between them automatically by calling the parameterless procedure `TRANSFILE`. This procedure copies bytes from one device to another from the currently selected input channel to the currently selected output channel, until an end-of-file status is raised on either the input or output channel.

NOTE: – in `INPUT`, this procedure works only for channel 0 and channels in the range 2-15 (i.e. not for `TTY` channels nor for logical channels). – in `OUTPUT`, this procedure works only for channels in the range 1-15 (i.e. not for logical channels).

Example:

```
COMMENT: THIS PROGRAM COPIES ONE FILE TO ANOTHER;
BEGIN
    INPUT(3,"DSK");
    OPEN.FILE(3,"from.dat");
    SELECT.INPUT(3);
    OUTPUT(7,"DSK");
    COMMENT: CREATE A VOID FILE;
    CREATE.FILE("to.dat");
    OPEN.FILE(7,"to.dat");
    SELECT.OUTPUT(7);
    TRANSFILE;
    CLOSE.FILE(3,7);
```

```
END.
```

16.12. CURRENTLY SELECTED CHANNEL NUMBERS

The number of the channel currently selected for input or output may be obtained by use of the integer parameterless procedures `INCHAN` or `OUTCHAN`.

E.g.

```
WRITE("CHANNEL OF INPUT =");PRINT(INCHAN);NEWLINE;
WRITE("CHANNEL OF OUTPUT=");PRINT(OUTCHAN);NEWLINE;
```

17. THE OPERATING ENVIRONMENT

taxi has all the procedures enlisted in the operating environment of DEC system-10/20 ALGOL and the ISO 1538-1984 documents.

In the following, they are grouped in the same sub-chapters of the DEC system-10/20 ALGOL user manual.

17.1. MATHEMATICAL PROCEDURES

Undoubtedly, ALGOL is a math-driven language. Its power as an algorithm-maker is shown by the Algol Bulletin, that was published from March 1959 to August 1988. The algorithms presented in the Algol bulletin prove that ALGOL had a great impact on the scientific and mathematical world, because of its structured programming.

So, I extended taxi mathematical abilities, freeing the programmer from designing common useful math functions, without wasting time for designing them.

The following procedures in general expect one real numeric argument and yield a real numeric result. If a literal numeric value or mathematical expression is given as argument, it is interpreted (or converted) to a real number. All trigonometric functions work with radians. The function `INTEGRAL` is more complex and follows its own rules.

`ABS` (Absolute value)
Domain: $x \in \Re$
Range: $y \in \Re \mid y \geq 0$

`ARCCOS` (Arc cosine)
Domain: $x \in \Re \mid -1 \leq x \leq 1$
Range: $y \in \Re \mid 0 \leq y < \pi$

`ARCCOSEC` (Arc cosecant)
Domain: $x \in \Re \mid x \leq -1 \cap x \geq 1$
Range: $y \in \Re \mid -\pi/2 < y < \pi/2$

`ARCCOSECH` (Arc hyperbolic cosecant)
Domain: $x \in \Re \mid x \neq 0$
Range: $y \in \Re \mid y \neq 0$

`ARCCOSH` (Arc hyperbolic cosine)
Domain: $x \in \Re \mid x \geq 1$
Range: $y \in \Re \mid y \geq 0$

`ARCCOTAN` (Arc cotangent)
Domain: $x \in \Re$
Range: $y \in \Re \mid -\pi/2 < y \leq \pi/2$

`ARCCOTANH` (Arc hyperbolic cotangent)
Domain: $x \in \Re \mid -1 \leq x \leq 1$
Range: $y \in \Re$

`ARCSEC` (Arc secant)
Domain: $x \in \Re \mid x \leq -1 \cap x \geq 1$
Range: $y \in \Re \mid 0 \leq y < \pi$

ARCSECH (Arc hyperbolic secant)
Domain: $x \in \Re \mid 0 < x \leq 1$
Range: $y \in \Re \mid y \geq 0$

ARCSIN (Arcsine)
Domain: $x \in \Re \mid -1 \leq x \leq 1$
Range: $y \in \Re \mid -\pi/2 < y < \pi/2$

ARCSINH (Arc hyperbolic sine)
Domain: $x \in \Re$
Range: $y \in \Re$

ARCTAN (Arctangent)
Domain: $x \in \Re$
Range: $y \in \Re \mid -\pi \leq y \leq \pi$

ARCTANH (Arc hyperbolic tangent)
Domain: $x \in \Re \mid -1 \leq x \leq 1$
Range: $y \in \Re$

COS (Cosine)
Domain: $x \in \Re$
Range: $y \in \Re \mid -1 \leq y \leq 1$

COSH (Hyperbolic Cosine)
Domain: $x \in \Re$
Range: $y \in \Re \mid y \geq 1$

COSEC (Cosecant)
Domain: $x \in \Re$
Range: $y \in \Re \mid y \leq -1 \cap y \geq 1$

COSECH (Hyperbolic cosecant)
Domain: $x \in \Re \mid x \neq 0$
Range: $y \in \Re \mid y \neq 0$

COTAN (Cotangent)
Domain: $x \in \Re \mid \pi < x < 0$
Range: $y \in \Re$

COTANH (Hyperbolic cotangent)
Domain: $x \in \Re \mid x \neq 0$
Range: $y \in \Re \mid y < -1 \cap y > 1$

ENTIER (Largest non-exceeding integer)
Domain: $x \in N$
Range: $y \in N$

`ERF` (Error function)
It is defined as

$$\texttt{ERF(x)} = \text{erf(x)} = \frac{2}{\sqrt{\pi}} \int_0^x e^{(-t*t)}\, dt$$

Domain: $x \in \Re + \mid x > 0$
Range: $y \in \Re \mid 0 < y < 1.0$

`EXP` (Exponential function)
Domain: $x \in \Re$
Range: $y \in \Re \mid y \geq 0$

`GAMMA` (Gamma function)
It is defined as

$$\texttt{GAMMA(t)} = \Gamma(t) = \int_0^\infty x^{t-1} e^{-x} dx$$

Domain: $x \in \Re \mid x \neq -k,\ k \in N \mid k \geq 0$
Domain: $t \in \Re \mid t > 0$
Range: $y \in \Re$

`INTEGRAL` (Finite Integral)
It is defined as

$$\texttt{INTEGRAL("f(x)",a,b,x [,n])} = \int_a^b f(x) dx$$

Domain: $x \in \Re$ (*range depends on the expression*)
Domain: $y \in \Re$ (*range depends on the expression*)

The function $f(x)$ is any mathematical ALGOL expression in the form of a string, with the integration variable occurring in it (otherwise it is a constant integration). The function must be continuous in the integration interval. The variable of integration must exist. Any other variables occurring in the expression stand for themselves and act as constants. The integration variable is written in naked form, as a proper variable name. The value of *n*, which is optional, specifies the number of subdivisions expressed as power to 10. If not given, the value 5 is set as default (that is, $10^5 = 100'000$ subdivisions), which bears at least 8 precise and fast decimal digits[14]. All arguments are passed by value, except the integration variable, which is passed by name, and retains the value of the last computed abscissa at the end of the calculation.

The solution is searched using the *iterated rule*, which tries to solve, in place of the exact solution, the approximate solution:

$$\int_a^b f(x)\, dx \approx \frac{b-a}{n}\left(\frac{f(a)}{2} + \sum_{k=1}^{n-1} f\left(a + k\,\frac{b-a}{n}\right) + \frac{f(b)}{2}\right)$$

In the directory `files/` of the installation folder of taxi you can find the program `elli.alg`, that calculates an elliptic integral of the first kind, the file `integral.alg`, that calculates a polynomial integral, and the file `sinint.alg` that calculates a sinusoidal integral.

[14] Warning: subdivisions indices higher than 7 increase considerably the computation time without increasing correspondingly the precision. (The more sums, the more round-off errors). Roughly, 5 means at least 8 fast precise decimals, 6 means at least 9 precise decimals (using more time), while 7 means at least 10 precise decimals (using a lot of time). The value 8 yields at least 11 precise decimals (which is good) but with unbearably long computation time.

If `INTEGRAL` is used into a procedure, and the integration variable is passed by name into the procedure, it is **this** variable inside the procedure that must appear in the integration function. For an example, see the program `integral_test.alg`, that calculates an integral using the intermediate procedure `integ()`.

`LN` (Natural Logarithm)
Domain: $x \in \Re \mid x > 0$
Range: $y \in \Re$

`SEC` (Secant)
Domain: $x \in \Re$
Range: $y \in \Re \mid y \leq -1 \cap y \geq 1$

`SECH` (Hyperbolic secant)
Domain: $x \in \Re$
Range: $y \in \Re \mid 0 \leq y \leq 1$

`SIN` (Sine)
Domain: $x \in \Re$
Range: $y \in \Re \mid -1 \leq x \leq 1$

`SINH` (Hyperbolic Sine)
Domain: $x \in \Re$
Range: $y \in \Re$

`SQRT` (Square Root)
Domain: $x \in \Re \mid x \geq 0$
Range: $y \in \Re \mid y \geq 0$

`TAN` (Tangent)
Domain: $x \in \Re \mid \pi / 2 < x < \pi / 2$
Range: $y \in \Re$

`TANH` (Hyperbolic Tangent)
Domain: $x \in \Re$
Range: $y \in \Re \mid -1 < y < 1$

The procedures `ENTIER`, `ABS` and `SIGN` were detailed in Section 5.1.2.

The following procedures (belonging to the DEC system-10/20 ALGOL) expect one long real numeric argument and yield a long real numeric result. If a literal numeric value or mathematical expression is given as argument, it is interpreted (or converted) to a long real number. All trigonometric functions work with radians.

`LABS` (Absolute value)
Domain: $x \in \Re$
Range: $y \in \Re \mid y \geq 0$

`LARCTAN` (Arctangent)
Domain: $x \in \Re$
Range: $y \in \Re \mid -\pi \leq y \leq \pi$

LCOS (Cosine)
Domain: $x \in \Re$
Range: $y \in \Re \mid -1 \leq y \leq 1$

LEXP (Exponential function)
Domain: $x \in \Re$
Range: $y \in \Re \mid y \geq 0$

LLN (Natural Logarithm)
Domain: $x \in \Re \mid x > 0$
Range: $y \in \Re$

LSIN (Sine)
Domain: $x \in \Re$
Range: $y \in \Re \mid -1 \leq x \leq 1$

LSQRT (Square Root)
Domain: $x \in \Re \mid x \geq 0$
Range: $y \in \Re \mid y \geq 0$

The following conversion procedures are available (BOOL and INT, the only ones belonging to the DEC system-10/20 ALGOL, were detailed in Section 5.6):

BOOL (Convert to Boolean type)
Domain: $x \in \Re$
Range: $y \in N \mid y = -1 or 0$

DEGREES (Convert to degrees)
Domain: $x \in \Re$
Range: $y \in \Re$

INT (Convert to integer type)
Domain: $x \in \Re$
Range: $y \in N$

RADIANS (Convert to radians)
Domain: $x \in \Re$
Range: $y \in \Re$

TOLONG (Convert to long real type)
Domain: $x \in \Re$
Range: $y \in \Re$

TOREAL (Convert to real type)
Domain: $x \in \Re$
Range: $y \in \Re$

17.2. STRING PROCEDURES

For details of the string procedures CONCAT, LENGTH, SIZE, COPY, NEWSTRING, CONVERT, HEAD, TAIL, TAKE, DROP and the procedure DELETE, see Paragraph 13.7.

17.3. UTILITY PROCEDURES

17.3.1. Array Dimension Functions

The integer procedure DIM, which takes as parameter the name of an array of any type, yields a result that is the number of dimensions of the array. This is most useful when the user passes an array as a parameter and wishes to check if it is, for example, a matrix.

The integer procedures LB and UB also take as first parameters the name of an array; the second parameter is the subscript number (1 for the first dimensions, 2 for the second dimension and so on). The result is the lower or upper bound, respectively, of the subscript specified by the second parameter. The following procedure uses these to clear real square matrices:

```
PROCEDURE ZERO(A); ARRAY A;
BEGIN
   INTEGER I,J;
   IF DIM (A) = 2 THEN BEGIN
      INTEGER L1,L2,U1,U2;
      L1 := LB(A,1); U1 := UB(A,1);
      L2 := LB(A,2); U2 := UB(A,2);
      FOR I := L1 UNTIL U1 DO
         FOR J := L2 UNTIL U2 DO
            A[I,J] := 0;
   END
END ZERO;
```

17.3.2. Minima and Maxima Functions

The integer procedures IMIN and IMAX, the real procedures RMIN and RMAX, and the long real procedures LMIN and LMAX are used, respectively, to determine the minimum or maximum of a series of arguments of the appropriate type. These procedures normally accept any number of parameters that may be contained in the input line.

For example:

```
I := IMIN(J,K,4,L);
X := RMAX(Y+Z,RMIN(Y-Z,Q);
```

NOTE: the IMIN and IMAX expect an integer values list, and implicitly convert any not integer value in the list to the integer type.

17.3.3. Field Manipulation

The procedures GFIELD and SFIELD enable the user to manipulate a field within an integer number, literal or contained into a variable. The integer parameters I and J specify a group of bits of length J whose rightmost bit is the I'th bit (counting from zero at the right-hand side). The byte specified varies from 1 to 31 in length and may be at any position in the variable. I may range from 0 to 30, with the constraint I + J <= 31. If values don't fit these rules, they are shaped accordingly.

The integer procedure GFIELD uses I and J as the second and third parameters; the first parameter is the variable or the constant containing the base value. The returned result is the value of the group of bits (right justified) specified by I, J. Thus:

```
INTEGER A;
A:=117;
PRINTLN(GFIELD(A,2,4));
```

returns the value of the group of bits from 2 through 5 of the integer A:

```
          13
```

(Being 117=1110101 in binary, with bit 0 on the right, the statement returns bits 2,3,4,5 and thus the sequence 1101, which is 13.)

The typeless procedure `SFIELD` sets a byte specified by the second and third parameters `I`, `J` to the value specified by the fourth parameter, of type integer. Thus

```
INTEGER A;
A:=117;
SFIELD(A,2,4,0);PRINTLN(A);
```

zeros the group of bytes 2,3,4,5 specified in the first example, returning

```
          65
```

(The statement sets to null bits 2,3,4,5 and thus the number becomes 1000001=65.) The statement:

```
SFIELD(A,2,4,7);PRINTLN(A);
```

sets bits 2,3,4,5 to the bits of 7=0111, returning

```
          93
```

(The number 1110101 becomes 1011101=93.)

If the fourth argument in the list is greater in bits-depth than the required group of bits, only the part that fits is considered, starting from its rightmost bit. E.g.

```
SFIELD(A,2,4,64);PRINTLN(A);
```

tries to overwrite bits 2,3,4,5 of number in A, using the number 64, which is in binary 64=100000; since the four rightmost bits are used, that is 0000, the effect is the same as using zero; the result is:

```
          65
```

NOTE: the AA-0196C-TK ALGOL manual provided instructions that were left-driven, while the instructions given here are right-driven, as used by current literature. The result is the same, as long as you count the `I`'th bit starting from bit 0 on one side, for a length `J` of bits counting towards the other side;

```
      right-driven:  1110101
                     6543210
      left-driven:   1010111
                     0123456
```

This is a minor deviation from the DEC system-10/20 ALGOL.

17.4. DATA TRANSMISSION PROCEDURES

For details of these procedures refer to Chapter 16.

17.5. INTERFACE PROCEDURES

The DEC system-10/20 ALGOL had a set of procedures enabled to invoke an external Fortran procedure (predeclared with the `EXTERNAL` statement), which could or not have a return value; the following procedures were provided:

- `CALL` (untyped)
- `ICALL` (integer type)
- `RCALL` (real type)
- `DCALL` (double-precision Fortran type)
- `LCALL` (logical Fortran type)

These had to be used with Fortran version F-40; if a Fortran version F-10 had to be used, the same procedure names were appended to the "F10" token, thus they were:

- `F10CALL` (untyped)
- `F10ICALL` (integer type)
- `F10RCALL` (real type)
- `F10DCALL` (double-precision Fortran type)
- `F10LCALL` (logical Fortran type)

In taxi, only one version of `CALL` is enabled, with different usage. The procedure

```
CALL(X);
```

invokes another instance of taxi over the program `X`, where `X` is a proper string (literal or variable) containing the name of the ALGOL textual program, along with its path if necessary. If the `".alg"` extension is not specified it is eventually attached to the name.

The execution of the called program starts at the same position of the `CALL` statement, and thus its output is interspersed with the output of the caller.

An error in the called program prints an error message (as usual) but the caller is not stopped. `CALL` provides a syntax that can retrieve the occurred error code. If the syntax

```
CALL(X,N);
```

is used, the return status of the called program is stored in `N`, for the user evaluation of the results (and to treat possible errors). The exit status follows the following conventions:

- a zero means a regular execution (no errors).
- a negative value means a UNIX error (here not quantified, because it depends on the various UNIX versions). This error is not an ALGOL error emitted by taxi, but something returned by the O.S.
- a positive value is the ALGOL error code, identifying what happened in the called program.

17.6. GENERAL INFORMATION ROUTINE

The integer procedure `INFO`, depending on the value of the parameter specified, provides information about various aspects of the environment.

Parameter	Scope
0	program size in bytes
1	date
2	time (seconds since midnight)
3	time (milliseconds since midnight)
4	runtime (milliseconds)
7	interpreter version

Other values simply return zero. All arguments are integer numeric values; returned values are generally of integer type, apart for `INFO(7)`. (See ahead.)

- `INFO(0)` returns the real dimension in bytes of the running program (not the meta-language generated by the pre-parser), and including the loaded libraries; thus, it may be bigger than the value returned by the shell.

- `INFO(1)` returns the current date in compact format; the date is returned as an integer in the form `yyyymmdd`, with zero-padded numbers. (E.g. 20160502 for May 2, 2016.)

- For what about `INFO(2)`, `INFO(3)` and `INFO(4)`, returned values are based upon time as calculated by the underlying machine; so, they cannot be absolutely precise, because they are retrieved by the ticks (i.e. CPU cycles) of your machine.

- `INFO(7)` returns a real number, in the form `v.MMmmm`; the integer part is the version number, the two following decimals are the major version number and the three following decimals are the minor version number (for instance, 1.02004 is returned for version 1.02.004); `INFO(7)` is suitable to be printed with `PRINT(INFO(7),1,5).`

Other examples:

```
PRINT(INFO(4)) ;
```

might produce

```
1134600
```

the job's runtime up to now in milliseconds, whereas

```
PRINT(INFO(5));
```

simply returns zero.

17.7. EXECUTION TIMING

To measure the execution timing, option `-t` comes in handy, but the measuring of time performed by this option includes the pre-parsing and the closing operations, that are executed by taxi for the startup and the analysis of the source, so that it's surely greater than the actual execution runtime of your ALGOL program[15].

To record the execution timing in more steps, including only the parts of code you want to register, you can use the procedure `CLOCK`, which returns in milliseconds the clock since taxi's start. The returned value is shortened by the argument value, which is interpreted as an integer. The first time `CLOCK` is invoked, it should be stored in a variable as the time-zero value:

```
N = CLOCK(0)
```

Any subsequent usage of this procedure with argument `N` will return the distance in milliseconds from the time-zero value, for instance:

```
PRINT(CLOCK(N))
```

This technique measures the progression of the ALGOL execution between the two program lines.

[15] To be honest, the startup phase takes a very small amount (few milliseconds), but this varies depending on the quantity of `BEGIN-END`, `IF-THEN-ELSE`, `WHILE-DO` and `FOR-DO` structures found. In any case, `CLOCK` was built to return possibly the most precise value.

Actually, the following expression holds:

```
CLOCK(N) = CLOCK(0) - N
```

The procedure CLOCK comes from Algol-20 (the ALGOL created by the Carnegie Institute of Technology in Washington in 1965), adapted for taxi to treat milliseconds rather than seconds.

Another technique involves two independent measures:

```
START := CLOCK(0);
...
... <anything else>
...
FINISH := CLOCK(0);
WRITE("THE PROCESS TOOK "); PRINT(FINISH-START);
WRITE(" MILLISECONDS[N]");
```

in a more classic way.

17.8. DATE AND TIME IN ASCII FORMAT

Three routines are provided for returning the current time and date in string format suitable for printing without modification. The two date routines, FDATE and VDATE, give the option of a standard three-character abbreviation for the month (FDATE), or a variable-length string with the name of the month in full (VDATE). In both cases the year is given in full. String procedure TIME gives an eight-character string with the current time as HH:MM:SS.

For example

```
WRITE (VDATE) ; NEWLINE; WRITE(TIME);
```

would produce an output as in

```
05-JANUARY-2021
12:16:55
```

whereas

```
WRITE (FDATE) ; NEWLINE; WRITE(TIME);
```

would produce an output as in

```
27-JUL-2024
14:18:24
```

NOTE: the month string is always in capital letters.

17.9. RANDOM NUMBER ROUTINE

Three routines have been included to provide a random number generation capability. The number generator RAND is similar to that used in the DEC system-10/20 BASIC library. The floating-point output number is distributed uniformly between 0 and 1.

The taxi version, unlike BASIC and like the DEC system-10/20 ALGOL, is initialized randomly. If a repeatable sequence of pseudo-random numbers is required, then the procedure SETRAN should be called before the first call to RAND, with a fixed initial value, for instance:

```
SETRAN(-1)
```

should be included in the ALGOL program. If the argument is zero, the call to SETRAN calculates a new seed based on current time (i.e. performs user randomization).

The third procedure is SAVRAN, which returns the value of the last generated random number without invoking the number generator.

RAND and SAVRAN are real procedures, SETRAN is non-type.

17.10. Pausing a program

The typeless and parameterless procedure PAUSE merely exits to the underlying shell, in such a way as to allow execution to be continued by typing, from the shell itself, the exit command[16]. This is provided to allow, for example, a device to be assigned or to ascertain if a file exists, or whatever is the need that cannot be obtained by the program itself.

To signal the start of pausing, the message

```
Paused. Type 'exit' to restore.
```

appears. When exit is typed, the messages

```
exit
Restoring the program.
```

appear. All these messages separate the ALGOL output from the output of the shell.

17.11. PROGRAM TRACING AND DEBUGGING

taxi makes available two typeless and parameterless procedures, ONTRACE and OFFTRACE that respectively enable and disable the dynamic tracing of procedures executions. When ONTRACE and OFFTRACE are invoked, their own debug line is not printed, but a system string appears. E.g.

```
ONTRACE;
...
...
OFFTRACE;
```

Produces the effect:

```
<Trace enabled>

[...]

[...]

<Trace disabled>
```

See chapter 19 for the complete exposure of the debug features.

[16] The underlying environment in the original DEC manual was called *Monitor*, which required the user to type CONTINUE to make execution go on; since taxi is a Unix program, it falls back to the shell, and to exit from a Unix shell, the exit command is required.

17.12. VARIABLE CONTENT TRACING

The typeless procedure DUMP lets the programmer know the content of the variables, to see if they contain what is expected. The DUMP output is directed always to the screen, regardless of the selected output channel. Syntax:

```
DUMP
DUMP(N)
DUMP ALL
DUMP var1, var2, var3....
```

DUMP accepts numerical values in parentheses or string arguments. In case of numerical argument, this is one integer parameter which represents the number of block-levels to be dumped, starting from the current level and towards the root level. A value of zero has the effect of dumping ALL.

The definition of a 'level' is the following:

1 any BEGIN starts a level, and its correspondent END closes that level; the current level is level 1, and all container levels (down to the root level) increase their level number hierarchically (being the root level the highest number).

2 any PROCEDURE, when executed, starts a level, in which argument variables and the typed return variable are declared; this level is closed when the runtime PROCEDURE process ends.

3 nothing else starts a level.

E.g.

```
DUMP(1)
DUMP
```

dump only current level, whereas

```
DUMP(3)
```

dumps the current level and the two consecutive lower levels.

```
DUMP(0)
DUMP ALL
```

dump all levels down to the root level. (The root level can be hard to be calculated manually, for long and complicated programs.)

The procedure DUMP can also use string arguments; in this case, they are variables names, and DUMP shows their content; variable names are written in a list after DUMP, without brackets and separated by commas:

```
DUMP height,weight,....
```

As said, level 1 is the current level, which is the level where DUMP is found. The root level, layered some levels distant, has one specific level number, qualifying the "distance" between current level and the root level. Any variable which is declared in the space between the current and the root level and is visible for the current level at the time DUMP is invoked, is dumped.

The following program shows how DUMP can be used.

```
BEGIN
   INTEGER A,B,C,D,E;
   A:=B:=C:=D:=E:=24;
   BEGIN
      REAL F;
      LONG REAL G;
      F:=ln(A);
      G:=F+B;
      BEGIN
         BOOLEAN H;
         INTEGER I,J,K;
         REAL L,M,N;
         H:=F>0;
         N:=3;
         DUMP;
         END;
      END;
END.
```

The output is the table of the variables visible for current level, marked as 1:

```
Level:1  Name:H [#8], Type is Boolean, Value is -1.000000
Level:1  Name:I [#9], Type is integer, Value is 0.000000
Level:1  Name:J [#10], Type is integer, Value is 0.000000
Level:1  Name:K [#11], Type is integer, Value is 0.000000
Level:1  Name:L [#12], Type is real, Value is 0.000000
Level:1  Name:M [#13], Type is real, Value is 0.000000
```

If `DUMP` is changed to `DUMP ALL` (there are three levels):

```
Level:3  Name:A [#1], Type is integer, Value is 24.000000
Level:3  Name:B [#2], Type is integer, Value is 24.000000
Level:3  Name:C [#3], Type is integer, Value is 24.000000
Level:3  Name:D [#4], Type is integer, Value is 24.000000
Level:3  Name:E [#5], Type is integer, Value is 24.000000
Level:2  Name:F [#6], Type is real, Value is 3.178054
Level:2  Name:G [#7], Type is long real, Value is 27.178054
Level:1  Name:H [#8], Type is Boolean, Value is -1.000000
Level:1  Name:I [#9], Type is integer, Value is 0.000000
Level:1  Name:J [#10], Type is integer, Value is 0.000000
Level:1  Name:K [#11], Type is integer, Value is 0.000000
Level:1  Name:L [#12], Type is real, Value is 0.000000
Level:1  Name:M [#13], Type is real, Value is 0.000000
Level:1  Name:N [#14], Type is real, Value is 3.000000
```

All variables in the levels are shown. If `DUMP` is put in another block:

```
BEGIN
   INTEGER A,B,C,D,E;
   A:=B:=C:=D:=E:=24;
   BEGIN
      REAL F;
      LONG REAL G;
      F:=ln(A);
```

```
      G:=F+B;
      BEGIN
         BOOLEAN H;
         INTEGER I,J,K;
         REAL L,M,N;
         H:=F>0;
         N:=3;
      END;
      DUMP ALL;
   END;
END.
```

the output would be:

```
Level:2  Name:A [#1], Type is integer, Value is 24.000000
Level:2  Name:B [#2], Type is integer, Value is 24.000000
Level:2  Name:C [#3], Type is integer, Value is 24.000000
Level:2  Name:D [#4], Type is integer, Value is 24.000000
Level:2  Name:E [#5], Type is integer, Value is 24.000000
Level:1  Name:F [#6], Type is real, Value is 3.178054
Level:1  Name:G [#7], Type is long real, Value is 27.178054
```

The `DUMP` mask is read as follows:

- `Level`: the level number; the current level (where `DUMP` is called), is level number 1; next levels are thus numbered 2,3,...,N; lateral levels are not seen at all, being out of reach (see ahead). The highest level is the root level.
- `Name`: the variable name, followed by the variable index in square brackets. There is one index for each variable.
- Type: the type of the variable.
- Value: the variable content.

If `DUMP` is needed only for specific variables:

```
BEGIN
   INTEGER A,B,C,D,E;
   A:=B:=C:=D:=E:=24;
   BEGIN
      REAL F;
      LONG REAL G;
      F:=ln(A);
      G:=F+B;
      BEGIN
         BOOLEAN H;
         INTEGER I,J,K;
         REAL L,M,N;
         H:=F>0;
         Name:=3;
      END;
      DUMP B,C;
   END;
END.
```

The output is

```
Level:2  Name:B [#2], Type is integer, Value is 24.000000

Level:2  Name:C [#3], Type is integer, Value is 24.000000
```

with the usual mask interpretation and the correspondent level. If a wrong call is used, for example with `DUMP K` instead of `DUMP A`, a message would appear:

```
taxi runtime processor error in the program at line 16:
Illegal variable or variable out of scope for DUMP.
DUMP K;
     ^
Error code 62
```

This is because variable `K` exists only when the relative `BEGIN` is entered and its level created and is destroyed when the level is terminated, thus `DUMP` cannot see it.

If you happen to have a variable called `ALL`, invoking `DUMP ALL` is not interpreted as "dump variable `ALL`" but "dump all variables". To solve this interpretation, put it not as first by using a comma:

```
DUMP ,ALL
```

18. ISO 1538-1984 (E) INTEGRATION AND COMPATIBILITY

The ISO 1538-1984 specifies, in Appendix 1, that two subsets of ALGOL 60 are defined, denoted as level 1 and level 2, if level 0 is the language version with the full architecture.

The level 1 is defined as equal to level 0 but with the following restrictions:

1 The OWN declarator is not included.

2 Additional restrictions are placed upon actual parameters as given by the following replacement lines to the table in Section 4.7.5.5

Formal parameter	Mode	Actual parameter
integer	name	integer expression (see 4.7.5.2)
real	name	real expression (see 4.7.5.2)
integer array	value	integer array (see 4.7.5.3)
real array	value	real array (see 4.7.5.3)
typeless procedure	name	typeless procedure (see 4.7.5.3)
integer procedure	name	integer procedure (see 4.7.5.3)
real procedure	name	real procedure (see 4.7.5.3)

3 Only one alphabet of 26 letters is provided, which is regarded as being the lower case alphabet of the reference language.

4 If deleting every symbol after the twelfth in every identifier would change the action of the program, then the program is undefined.

The level 2 subset consists of restrictions 1-3 of level 1 and also:

5 Procedures may not be called recursively, either directly or indirectly.

6 If a parameter is called by name, then the corresponding actual parameter may only be an identifier or a string.

7 The designational expressions occurring in a switch list may only be labels.

8 The specifiers switch, procedure and (type) procedure are not included.

9 A left part list may only be a left part.

10 If deleting every symbol after the sixth in every identifier would change the action of the program, then the program is undefined.

taxi has none of these restrictions, so I can state it is at least a level 0 ALGOL. Help me to prove it, by testing it under every aspect...

18.1. GENERAL

taxi is quite fully ISO 1538-1984 compliant, even if some decisions in favour of the DEC system-10/20 ALGOL were taken when the two differentiated; this is due to the complete procedures DEC can offer, while the ISO 1538-1984 lacks some specific I/O statements. (For example, opening a file in input/output.)

In any case, the ISO conformance is guaranteed at least for the system procedures, the math environment and all the procedures in the Environmental Block, and all differences between taxi and the ISO 1538-1984 are always exposed, though scattered all over the entire manual, where appropriate.

Escape characters are correctly input and output, as loosely imposed in Section 2.6.3 of the Standard, and according to Section 18.7.1 of this manual.

In the following, all the ISO 1538-1984 features enabled in taxi are presented.

18.2. ISO INTEGER PROCEDURES

The following integer procedures are provided (see the ISO 1538, Appendix 2, page 14):

`ABS(E)`
absolute value $|E|$ (real argument, real result)

`IABS(E)`
absolute value $|E|$ (integer argument, integer result)

`SIGN(E)`
sign of the real argument

`ENTIER(E)`
largest integer not greater than `E` (real argument, integer result)

18.3. ISO MATHEMATICAL FUNCTIONS

The following math real functions are provided (see the ISO 1538 Appendix 2, page 14):

`SQRT(E)`
square root $\sqrt{\mathrm{E}}$ (error message differs)

`SIN(E)`
sine of `E` expressed in radians

`COS(E)`
cosine of `E` expressed in radians

`ARCTAN(E)`
principal value returned in radians of the arctangent of `E`

`LN(E)`
natural logarithm of `E` (error message differs)

`EXP(E)`
exponential function of `E` (error message differs)

Besides, the following ISO 1538-1984 functions are provided (see Section 3.3.4.3 page 7):

`EXPI(I,J)`
exponentiation $\mathtt{I}^{\mathtt{J}}$ (`I` and `J` integer values)

`EXPR(X,Y)`
exponentiation $\mathtt{X}^{\mathtt{Y}}$ (`X` and `Y` real values)

`EXPN(R,I)`
exponentiation $\mathtt{R}^{\mathtt{I}}$ (`R` real and `I` integer)

18.4. ISO TERMINATING PROCEDURES

The following procedures are provided (see the ISO1538 Appendix 2, page 14):

`STOP`

anticipated end of program

NOTE: the `STOP` procedure executes a brutal and immediate stop of the execution; none of the existing structures (`IF-THEN`, `WHILE`, `FOR` and `BEGIN` blocks) are closed, and opened channels or files are forced close by the ending routine. The execution terminates with the message:

```
taxi execution interrupted by STOP at line <n>
```

where `<n>` is the line where the `STOP` occurs.

`FAULT(STR,R)`

error message printing and anticipated end

NOTE: error treatment in taxi is built according to the DEC system-10/20 ALGOL behaviour[17]. But the ISO standard included a procedure for treating runtime user errors (though this could be also accomplished by a proper procedure using only `WRITE` or `OUTSTRING`). So I introduced the ISO `FAULT`, which has at least the remarkable advantage of uniforming the printing of error messages, and may reveal effectively helpful in devising personal error treatment routines.

E.g.

```
FAULT("error in argument",Y);
```

(supposing `Y` contains some specific error-prone value that violates some conditions), produces

```
fault  error in argument  -3.000000
```

The aspect of the error message is totally ISO 1538-1984-compliant (respecting spaces and the lower-case characters).

In designing the `FAULT` procedure, I thought it could be improved to include the DEC side as well; therefore, in taxi, `FAULT` has two other captivating features.

The first makes optional the second argument; that is, if the second argument is not given, only the textual message is printed. E.g.

```
FAULT("error in argument");
```

produces

```
fault  error in argument
```

The error string may be a literal value or contained into a string variable or string array element.

The second improvement makes `FAULT` accept a numeric value as argument; in this case, it is interpreted as the error code of the taxi error list[18]. This can uniform the error messages with DEC. E.g.

[17] I must warn that the error message list in the DEC manual is largely incomplete, so I had to design personal and specific error messages to help the programmer to understand what and where the error occurred.

[18] type `taxi --error-list` to see the whole list on the screen.

```
FAULT(47)
```

prints

```
fault  Division by zero
```

In this way, FAULT becomes the *trait d'union* of the two worlds, at least for what about the printing of error messages.

18.4.1. Special Error Testing Features

A special error-test feature is the *zero-condition*: error condition 0 (zero) is a harmless error condition that is never returned by taxi, and that can be conveniently used by FAULT to specify an error condition test without having to build a real error in the code.

I must warn you though that, since taxi is a UNIX program, it uses the convention that the returned value 0 (zero) is normally considered by the shell as the code for *EXIT with SUCCESS* (in UNIX terminology); this means that, in using the procedure CALL (see 17.5), such a case is received as a regular ending by the called program, and if you meant to catch the zero error condition this would be undistinguishable from a regular execution and thus useless.

Take this into account when using FAULT within CALL calls.

18.5. ISO INPUT/OUTPUT PROCEDURES

I have anticipated earlier that the ISO 1538-1984 offers no procedures for establishing a connection to a file or a device (leaving the responsibility to the implementer); anyway, the following input/output procedures described in the ISO 1538-1984 are provided (see Appendix 2, page 14-16):

`INCHAR(channel,str,int);`
: Set `int` to the value corresponding to the first position in `str` of current character on `channel`. Set `int` to zero if character not in `str`. Move `channel` pointer to next character. The string `str` can contain escape characters in the form `\x`, that will be correctly evaluated as such.

`OUTCHAR(channel,str,int);`
: Pass to `channel` the character in `str`, whose position corresponds to the value of `int`. The first character of the string `str` has position 1.

`LENGTH(str);`
: Return the number of characters in the open string `str` enclosed by the outermost string quotes, after performing any necessary concatenation as defined in the ISO Standard at Section 2.6.3 and also `LENGTH` in Section 13.7.2 of this manual.

`OUTSTRING(channel,str);`
: Pass to channel all characters of `str`, until the end of the string. It is a wrapper for `OUTCHAR`. Even if not explicitly specified in the ISO norm (that only asserts in 2.6.3 the need for an escaping sequence for the string delimiters), `OUTSTRING` was enabled to parse all the escape characters specified in 18.8.1.

`OUTTERMINATOR(channel);`
: Output a terminator for use after a number to `channel`. In taxi the terminator is a space character.

`ININTEGER(channel,int);`
: `int` (which must be a proper integer variable) takes the value of an integer, as defined in Section 2.5.1 , read from `channel`. The terminator of the integer may be either a space, a newline or

a semicolon. Any number of spaces or newlines may precede the first character of the integer representation.

`OUTINTEGER(channel,int)`

Pass to `channel` the characters representing the value of `int`, followed by a terminator. In the case of a negative number, the output is preceded by the minus sign.

`INREAL(channel,re);`

`re` takes the value of a real number read from `channel`. Blanks may follow the exponent symbol, which can be `&`, `@` or `E`. The terminator of the integer may be either space, a newline or a semicolon. Any number of spaces or newlines may precede the first character of the real representation.

`OUTREAL(channel,re);`

Pass to `channel` the characters representing the value of real `re`, followed by a terminator. In the case of a negative number, the output is preceded by the minus sign. The exponential form is activated only when the number cannot fit the 10 characters available space[19] exponential excluded. The output format is formed by the integer part plus the decimals, in total 10 characters, with the eleventh position occupied by the dot, and three or four more characters for the positive or negative exponential part.

18.6. ISO ENVIRONMENTAL ENQUIRIES

NOTE: the environmental enquiries exposed in ISO1538 Appendix 2, page 16 , quote "*maxreal, minreal, and maxint are, respectively, the maximum allowable positive real number, the minimum allowable positive real number, and the maximum allowable positive integer, such that any valid expression of the form*

```
<primary> <arithmetic operator> <primary>
```

will be correctly evaluated, provided that each of the primaries concerned, and the mathematically correct result lies within the open interval (-maxreal, -minreal) or (minreal, maxreal) or is zero if of real type, or within the open interval (-maxint, maxint) if of integer type..."

So, the following environmental enquiries, contained in the ISO1538-Appendix 2, page 16) are provided (see also `taxi.h`)

`MAXREAL`

The maximum positive allowed real number.
It is set to 1E+16.[20]

`MINREAL`

The minimum allowed negative real number.
It is set to 1E-16.

[19] The ISO `outreal` at page 16 of the Standard uses a formula to retrieve the width of the printing number, that is the precision for the decimals. The formula is

$$entier(1.0 - \frac{ln(epsilon)}{ln(10.0)})$$

Now, using a normal value as 1E-15 for an average `epsilon`, we get 16 as depth. (Even more, if you think that nowadays machines are capable of holding even more digits.) Probably it was intended as the total characters space for integer part, decimals and the exponential part. This turns out to be a little strange, if you consider that the ISO `outreal` procedure prints *all the digits*, zeros included. So I chose to retain 10 digits (exponential part excluded) as the total digits width for `OUTREAL`, basing on the same output of the DEC protocol for the decimal part, and add the exponential only when due. This is a minor difference with the ISO document since `OUTREAL` in taxi is really ISO-compliant, except for the width.

[20] This limit is valid only for the ISO procedure `OUTREAL`. In case a `REAL` number is printed with `PRINT`, the limit of computation is not `MAXREAL` but `MAXLONGREAL` (see ahead).

`MAXINT`
The maximum computable integer value.
It is set to 2147483647.

`EPSILON`
The smallest positive real number such that

```
1.0+EPSILON > 1.0
1.0 > 1.0-EPSILON
```

It is set to 9.9999999999999999E-16.

In addition to the default ISO 1538-1984 environmental enquiries, taxi has also the following constants:

`INFINITY`
The number beyond the maximum computable positive real/long real.
It is printed as `inf` (not by taxi but by the gcc).

NOTE: `INFINITY` returns an 'infinite' number, which is not a proper number and is printed as `inf`. This has not to be confused with `INF`, which returns a computable number, the maximum positive real. `INFINITY` is the 'number' you get if you try to go past `INF`.

NOTE: if you try to assign `INFINITY` to an integer type, this will be assigned the value -2147483647, because the integer type has no infinite.

`INF`
Maximum computable positive real/long real.
It is set to 1.7E308.

NOTE: `INF` returns an 'infinitive', which is a proper number and is shown in full. `INF` and `MAXLONGREAL` can be confused because they are quite the same value. `INF` was introduced to set the 'true' limit of computation of the C double type. This has not to be confused with `INFINITY`, which is not a computable number.

`LONGEPSILON`
The smallest positive long real number such that

```
1.0&&0+LONGEPSILON > 1.0&&0
1.0&&0 > 1.0&&0-LONGEPSILON
```

It is set to 9.9999999999999999E-307.

`MAXLONGREAL`
Maximum allowed positive long real number.
It is set to 1.0E+308.

`MINLONGREAL`
Minimum allowed positive long real number.
It is set to 1.0E-308.

`MININT`
Maximum computable negative integer value.

It is set to -2147483648.

`PI;`
Greek π, equal to the ratio of a circle's circumference to its diameter. It is set to 3.14159265358979. `PI` returns a long real value.

For what about the limits previously exposed, remember what was said for the storing of real numbers[21]: since they are stored in the same C `double` memory space of the long real numbers, any computation involving real/long real numbers is not truncated to `MAXREAL` or `MINREAL`; the limits are only superimposed in printing (for use with `PRINT` and `OUTREAL`).

18.7. SPECIAL ISO-LIKE ADDITIONS

To gain perfect compatibility between the ISO 1538-1984 spirit and the DEC system-10/20 ALGOL procedures, and to reach a higher level of compatibility with other ALGOL environments, the following procedures, which don't belong to the DEC design or the ISO standard, are provided:

`INARRAY(channel,ar);`
`ar`, which is a regular numeric array, is filled with values read from `channel`. `INARRAY` was discussed in detail in Section 16.7.3.5 "Array Input/output".

`OUTARRAY(channel,ar);`
Array `ar` is printed to `channel` according to fixed rules. `OUTARRAY` was discussed in detail in Section 16.7.3.5 "Array Input/output".

`INSTRING(channel,str);`
`str`, which is a regular string variable, takes the value of the string read from `channel`; in case of a file line, `str` takes the value of the whole line up to END-OF-LINE; In case of input from the keyboard, `str` takes the value of the whole typed line up to the Enter key. Escape characters are correctly evaluated as such.

If the string is enclosed in double-quotes like `"this"` or in double single quotes like `''this''`, the surrounding quotes are not removed before assignment so that strings are retained as-is.

If `str` is successively assigned to a string variable of length `L`, only up to `L` characters of `str` are copied, and the rest is lost. Thus the user must take in account the length of the hosting string. (The usage of `NEWSTRING` may reveal useful in case one wants to enlarge the space of the default 255 characters hosting string, for instance to assign a file line, which may be longer.)

`INLONGREAL(channel,lre);`
`lre` takes the value of a number read from `channel`. Blanks may follow the exponent symbol, which can be `&&`, `@@`, `D` or `E`. If the exponential symbol is not doubled, taxi won't complain.

`OUTLONGREAL(channel,lre);`
Passes to `channel` the characters representing the value of the long real `lre`, followed by a terminator. The exponential form is activated only when the number cannot fit the 20 decimals available space. In the case of a negative number, the output is preceded by the minus sign.

`OUTBOOLEAN(channel,lbool);`
Passes to `channel` the words `false` or `true`, according to the truth value of the Boolean `lbool`, followed by a terminator.

[21] See paragraph 1.2.2.

`INCHARACTER(channel,char);`

`char` takes the value of an ASCII character read from `channel`. A character is a number in the range 0÷127. In case of input from keyboard, the character is read and not echoed, and no Enter key is required: after the keypress, the procedure ends.

`OUTCHARACTER(channel,char);`

Passes to `channel` the character number in the range 0÷127, as an ASCII character. No terminator follows.

`ERRC, ERRL`

These variables take respectively the error code number and the error line number in case of errors occurred in procedures that host a label as error return technique. (E.g. `INPUT`, `OUTPUT`, `OPEN-FILE`.) In case of normal usage, their both are set to -1.

Besides, the following command, which is a more general print statement, though not ISO-like, is provided:

`VPRINT(S);`

Prints to current output channel the string `S` (either literal or contained into a string variable or a string expression) followed by a New Line character. Interpretation of escape characters in 18.7.1 is provided. Any other escape used is interpreted literally. (E.g. `\w` is `w`.) The special string commands contained in square brackets (available in the DEC system-10/20 ALGOL `WRITE`) are here not supported and are interpreted literally.

18.7.1. Escape characters

Here are the escaping characters understood by taxi (and used in the ISO string procedures and also in the foreigner imported `VPRINT`):

```
alarm beep       \a
backspace        \b
form feed        \f
new line         \n
carriage return  \r
horizontal tab   \t
vertical tab     \v
backslash        \\
single quote     \'
```

In Unix, generally, the New Line character is performed by `\n`, while in the Mac world the same task is performed by `\r` and in DOS/Windows by `\r\n`; in any case, if you use `\n`, this will be correctly interpreted as a New Line escaping character in (hopefully) all the three environments.

18.8. DEC AND ISO 1538-1984: TWO DIFFERENT PHILOSOPHIES

DEC and ISO, while based upon the same ALGOL rules, bear a different conception in treating the input/output. There is good and bad in each world, as you will see, and you can use at any time the one you like best, or the one specifically targeted for your instance of output.

In the following, some specific differences are underlined.

18.8.1. DEC and ISO default channels

The main and most important difference between DEC and ISO I/O procedures (apart for the graphical issues) is that the DEC procedures `PRINT`, `WRITE`, `NEWLINE`, `READ` etc. (included `VPRINT`) act on the default input/output channels set by `SELECTINPUT` or `SELECTOUTPUT` (by default the `TTY` for both directions). That is, if you want to print on *another* channel, you have to open it and make it available.

The ISO 1538-1984 procedures, instead, like `INCHAR`, `OUTCHAR`, `OUTSTRING`, `ININTEGER`, `OUTINTEGER`, `INREAL`, `OUTREAL` do specify the channel in the argument queue. If the default `TTY` is to be used, the channel number to be specified is 0 (zero) for input and 1 (one) for output. Unfortunately, no means for opening other channels is provided by the ISO 1538-1984, so that you have to use the DEC ones for this purpose.

18.8.2. DEC and ISO numeric output

The main difference between numeric DEC output procedure and ISO output procedures is this:

- DEC `PRINT` procedures print numbers along with fixed rules: you can manoeuvre the output by deciding in advance the depth and width of numbers by changing the second and third arguments, and this gives great help when printing tables, and multiple results, matrices and so forth. A space character is always printed after a number, preceded by the minus sign if negative or a blank if positive.[22]

- ISO 1538-1984 `OUTREAL` and `OUTINTEGER` (and the apocryphal `OUTLONGREAL`), instead, print numbers more 'naturally'; after all, reading 25.40000000 is more comfortable than 2.540000000&+1; these procedures are designed to use the exponential form only when needed, while the DEC system-10/20 ALGOL prints real and long real numbers always in exponential form, unless you size the output on your own through `PRINT`.

18.8.3. DEC and ISO output line length

Another difference occurring between the two worlds managed by taxi is the way they manage the line length in the output:

– DEC `PRINT` and `WRITE` procedures keep track of the line counter, and if the printing position is past the value of 72, they print a New Line character, to avoid breaking up strings and numbers.
– ISO 1538-184 `OUTREAL`, `OUTINTEGER` and `OUTLONGREAL` procedures instead, while they keep track of the line counter, don't control if the printing position is past the screen limit. This adheres to the ISO documents (Environmental block, Appendix 2).

18.9. MANUAL CONVERSIONS FOR OTHER SOURCES

taxi does its best to convert foreign structures to proper ALGOL items, but certainly it cannot do everything. Certain features cannot be converted because they have a very special meaning for taxi, and thus must be changed manually, in particular, if an environment uses other channels for the TTY other than 0 (zero) or 1 (one), these have to be changed manually.

NOTE: Some specific statements cannot be converted. For instance, marst (an ISO ALGOL to C converter), uses instructions like `'print'` to output a list of items (numbers or strings) to channel 1 - the terminal - and `'inline'` to pass C-formatted strings to the C converter, which are supposed to be compiled along with the rest of the program. The first, `'print'`, has the same name of the `PRINT` procedure in the DEC system-10/20 ALGOL, which is targeted to print one number at each invocation, and thus cannot be converted (in the sense of DEC). The second, `'inline'`, requires the evaluation of the string, and case by case a new procedure (or procedures set) must be created to simulate the C effect. For instance, suppose

[22] It's curious noting that the SAIL program, available in the TOPS20 platforms didn't print a space *after* the number.

you have, into the body of a procedure called `retproc`:

```
inline("my.retval.u.real = rand();");
```

This is supposed to get some random value to the variable `my.retval.u.real`. To convert this in ALGOL, a manual similar effect can be obtained by using in place of the previous `inline` entry:

```
my.retval.u.real := RAND;
```

In all other cases, you first have to ascertain what is needed. Second, you have to find a trick that solves the problem.

19. DEBUGGING PROGRAMS

Being an interpreter, taxi has a simple debugger; it aims to help the programmer find where the error occurs, since error messages not always bring exact information.

The debugging is enabled in three ways: option `-d` enables the debug from the start since it is invoked on the command line. In the source, you can enable the debugging with the typeless procedure `ONTRACE` and can be disabled with `OFFTRACE`.

There are two acting debugger engines: the one acting on the pre-parsing phase and the one on the runtime phase. Let's examine them in detail.

Note: the line number and position number that the debugger writes are not those of your ALGOL original listing, but they refer to the tokenized and blanks-stripped version of the program.

19.1. THE PRE-PARSING PHASE

Option `-e` ('e' stands for 'extended debug') enables the debugging of the pre-parsing phase, and option `--verbose` increases the information given. During this phase, the following happens:

* Every `LABEL` is stored, removed from the code and printed in reverse colours.

  E.g.:
  ```
  Stored label #1 LAB1 (2222, 6, 27)
  ```

  The three numbers in brackets are 1) the numeric code associated to the label (unique for each unique label); 2) the line where `LAB1` is put; 3) the position in the line where `LAB1` is put. These values are used by `GOTO`, to know exactly where to jump.

* `BEGIN-END` couples are searched, and their relative positions are stored internally, for fast execution; a message in reverse colours is printed for each couple (in verbose mode only), like this:

  ```
  BEGIN [#1] at line 1 has:
  begin data: l=1, p=5
  end data  : l=22, p=0
  ```

  Each `BEGIN` must match one `END`. An error is raised at the first unmatched couple.

* Every `PROCEDURE` is analyzed for what about: start and end of `PROCEDURE` body, type of the `PROCEDURE` (the number in square brackets is the type code), number and type of arguments (if any), and the proper `BEGIN-END` main cycle is stored (if any); a message in reverse colours is printed for each `PROCEDURE` like this:

  ```
  PROCEDURE [#1] test at line 3 has:
  type = Real [20]
  arguments number = 2
    #1 X, real referenced variable
    #2 I, real valuated variable
  proc data : l=3, p=0
  begin data: l=5, p=0
  term data : l=8, p=3
  ```

The lines data (the last three) are printed only in verbose mode. Notice that a *referenced* variable is passed by name, a *valuated* variable is passed using the VALUE specification.

* Every IF-THEN and every IF-THEN-ELSE are searched and found; the parser stores the position of the start of the condition expression, the position of THEN (to call in case of true condition) and the position of ELSE (to call in case of false condition); this last may be omitted; the termination position is saved to make a quick jump to the end of the decisional part; a message in reverse colours is printed (in verbose mode only) for each IF-THEN-ELSE like this:

```
IF [#1] at line 12 has:
cond data: l=12, p=2
THEN data: l=12, p=9
ELSE data: l=17, p=4
term data: l=17, p=5
```

and for each IF-THEN like this:

```
IF [#1] at line 12 has:
cond data: l=12, p=2
THEN data: l=12, p=9
ELSE data: l=-1, p=-1
term data: l=12, p=12
```

where the ELSE clause is absent.

* Every FOR-DO and WHILE-DO section are searched and found; the parser stores the position of the variable code (or variables list) of the FOR or the condition expression of WHILE, and the termination position, to jump to when the cycle is over. The inner calculations are evaluated at run-time; a message in reverse colours is printed (in verbose mode only) for each FOR-DO and WHILE-DO like these:

```
FOR [#3] at line 14 has:
var data : l=14, p=3
term data: l=20, p=3

WHILE [#1] at line 11 has:
cond data: l=11, p=5
term data: l=18, p=3
```

* Finally, the pre-parse phase coordinates the PROCEDUREs, according to the field of application; for instance, if PROCEDURE A defines procedure B inside its body (not in the main body), the following appears in reverse colours:

```
Procedures categorizing:
set procedure B as subordinated to procedure A
```

This message informs you that B cannot be called from the main body or any other procedures but only from within A.

At the end of the pre-parsing phase, the runtime debugging is enabled automatically (option -d, the run-time debugger), so there's no need to call both.

19.2. THE RUNTIME PHASE

Option `-d` enables the debugging of the runtime phase. During this phase, the following happens:

* Every line of code under execution is printed in reverse colours and tokenized blanks-stripped form, enclosed in square brackets. The output, if any, is printed in the next line, otherwise a blank line appears. For instance the program lines:

```
INTEGER X4; X4:=24;
WRITE("X4  ="); PRINTLN(X4,4,2);
```

  show the following debug output:

```
[INTEGERX4]
Declared X4 [#1], Type is integer

[X4©24;]

[WRITE("X4  =")]
X4  =
[PRINTLN(X4,4,2);]
   24.00
```

  If on the same line lie two or more instructions, the debugger shows the whole line but debugs the current instruction only.

  The debug in itself ruins the output of the debugged program, because it clutters it with interspersed text. But after all, when everything gets smooth, the debug must not be invoked, so I judge this not relevant if the final scope is finding the mistake.

  This debug technique, even if naive and simple, is very helpful, because it lets you examine the program flow to understand why the program does not do what you intended.

  The various debug features are:

* Every variable declaration - even for a variable in the argument list of a `PROCEDURE` - shows its declarative assertion in reverse colours:

```
Declared X1 [#3], Type is real
Declared X2 [#4], Type is Boolean
Declared X3 [#5], Type is long real
Declared X4 [#6], Type is integer
Declared X5 [#7], Type is label
Declared X6 [#8], Type is string
```

  If a `PROCEDURE` with a return value was declared, the following appears in reverse colours:

```
Declared fictitious variable A [#1], Type is real
```

  The 'fictitious' adjective focuses on the fact that this is not a real variable, but it represents the return value of the `PROCEDURE`, with the same name of the `PROCEDURE`.

* Every assignment to the fictitious variable is shown:

```
 Assigned fictitious variable test [#1], Type is real, Value is
4.898979
```

* PROCEDURES must be executed when invoked, but they must be ignored in the calling level when met consecutively in the source; in this case the message in reverse colours appears:

```
 [REALPROCEDUREB;]
 Skipping procedure B
```

  This means: the line 'REAL PROCEDURE B' was found and skipped, and a jump to the first instruction following it was executed.

* IF..THEN..ELSE declarations behave a little differently; when IF is evaluated, only the condition is printed in reverse colour; the evaluation of the THEN..ELSE flag is done internally (and not reached by the debugger); then, on a different line the THEN *or* the ELSE part is printed, in reverse.

  E.g. this program piece

```
 IF A=0 THEN WRITE("DON'T DO")
 ELSE WRITE("DO");
```

  is debugged, in case of false condition (suppose A=1):

```
 [IFA=0]

 [WRITE("DO");]
 DO
```

  and is debugged, in case of true condition (A=0):

```
 [IFA=0]

 [WRITE("DON'T DO")]
 DON'T DO
```

  Summing up, the IF condition is printed, and then follows the right executed part (the THEN *or* the ELSE part).

  For the case where a BEGIN–END block follows THEN or ELSE, it is debugged as usual.

19.3. USING STEP-BY-STEP TRACING

Using the two procedures ONTRACE and OFFTRACE the programmer can enable the debugging on a specific part of the code, and not on the entire code. This second debugging technique shows the same lines of the classic debugger, but it stops at every procedure token (stop-mode), waiting for the programmer's intervention.

The commands available by the programmer are one-key characters (not echoed) that perform some specific actions:

Command H - help

By hitting the key **H**, the programmer will see a help screen that summarizes the commands of the debugger.

Command C - continue
By hitting the key **C**, the programmer will istruct the debugger to continue in debug mode, disabling the stop-mode; the execution will proceed until the end of the execution as if started with option `-d` (until the final `END` or until the next `OFFTRACE`).

Command G - go
By hitting the key **G**, the programmer will istruct the debugger to continue disabling the debug and the stop-mode as well; the execution will proceed until its natural end in normal mode.

Command X - exit
By hitting the key **X**, the programmer will instruct the debugger to stop immediately the execution. It's like a `STOP`, with the difference that the string

```
(terminated)
```

appears before the stop takes place, to signal that the execution was stopped by the programmer and not by the program itself. All closing procedures are anyway executed (file closing, memory freeing, etc.).

Command V - dump variable
By hitting the key **V**, the programmer will see a > at the start of the line to signal that he can input a variable name. This name is searched in the current context and, if found, variable is shown (as if `DUMP` was called upon it). If the variable does not exist or is out of scope, this is simply reported, with no effect on the execution.

Command P - dump procedure
By hitting the key **P**, the programmer will see a > at the start of the line to signal that he can input a procedure name. This name is searched in the current context and, if found, the procedure data are displayed as in option `-e` (option `--verbose` increases the information amount). If the procedure does not exist or is out of scope, this is simply reported, with no effect on the execution.

Any other key
By hitting any other key (space bar included) the debugger will step to the next token and will executed its procedure, along with the printing of the debug line.

In all a simple debugger that helps a lot, at least I think so.

20. IN THE END...

This program was designed, written and implemented just for fun. I'm not a professional programmer. But I like programming a lot. So here it is.

Infinite thanks to **Ian Jones**, a friend and a great tester, who compiled and ran taxi an incredibly high number of times, finding all sort of errors in the listings. He's been the mechanic of taxi!

Infinite thanks also to Bruce Axtens, who's always shown great willingness to help me; he is the master of the Windows world, the one who compiles my programs for Unix and makes them available for the Redmond operating system. He's a programmer, and he's able to spot bugs faster than a blink.

CAVEAT: *This text was written starting from the 1977 version of the DEC system-10/20 ALGOL programmer's Guide AA-O196C-TK, changing and updating the not complying parts, integrated with the ISO 1538-1984 features, quoting it from time to time. If owners of the copyrights of the cited documents should require any modification, I ask them to let me know, for the appropriated changes to be applied to this manual.*

This manual was written using gvim and it was type-processed using groff. If you find errors or if you don't like something, it's entirely my fault.

If you find any bugs, write to <ing dot antonio dot maschio at gmail dot com>, don't hesitate. I'll try to solve the bugs as soon as possible.

Two things are still to be said:

the first is that I hope you like taxi.

The second is: it's GPL, enjoy! ♥

Appendix 1: CONCEPTS INDEX

NOTE: in the following, index numbers refer to chapters, not page numbers; e.g. 2.3, 5.2.1

Appendix 2: LANGUAGE ELEMENTS INDEX

NOTE: in the following index:

- names in upper case bold characters denote reserved procedures and structures identifiers. E.g. **FOR BEGIN REAL**
- names in upper case italic characters denote non-reserved procedures and constants identifiers. E.g. *PRINT() OUTPUT() EPSILON*
- names in lower case bold characters denote math operators identifiers and symbols. E.g. **and eqv := < =>**
- index numbers refer to chapters, not page numbers. E.g. 2.3, 5.2.1

Table of Contents