NAME

ACK − Additional information on the Amsterdam Compiler Kit compilers

DESCRIPTION

These are the details on the Amsterdam Compiler Kit compilers for the languages C, Modula-2, and Pascal. The design decisions that were made where the respective standards allowed or mandated this, and the extensions that were implemented.

ANSI C REPORT

This section specifies the implementation-defined behavior of the ANSI-C compiler as required by ANS X3.159-1989.

ANS A.6.3.1

- Diagnostics are placed on the standard error output. They have the following specification:

  "<file>", line <nr>: [(<class>)] <diagnostic>

  There are three classes of diagnostics: 'error', 'strict' and 'warning'. When the class is 'error', the class specification is absent. The class 'strict' is used for violations of the standard which are not severe enough to stop compilation, for example the occurrence of non white-space after an '#endif' preprocessing directive. The class 'warning' is used for legal but dubious constructions, for example the declaration of a structure-tag in a parameter type list.

ANS A.6.3.2

- The function 'main' can have zero or two parameters. When it has two parameters, the first parameter is an integer specifying the number of arguments on the command line (including the command). The second parameter is a pointer to an array of pointers to the arguments (as strings).
- Interactive devices are terminals.

ANS A.6.3.3

- The number of significant characters is 64. Corresponding upper-case and lower-case letters are different.

ANS A.6.3.4

- The compiler assumes ASCII-characters in both the source and execution character set.
- There are no multibyte characters.
- There are 8 bits in a character.
- Character constants that cannot be represented in 8 bits are truncated.
- Character constants that are more than 1 character wide will have the first character specified in the least significant byte.
- The only supported locale is 'C'.
- A plain 'char' has the same range of values as 'signed char'.

ANS A.6.3.5

- The i80x86 and 68000 both have a two's complement binary-number system. Shorts are 2 bytes; ints are 2 bytes under 16-bits and 68000 MINIX-PC, 4 bytes under 32-bits ; longs occupy 4 bytes.
- Converting an integer to a shorter signed integer is implemented by ignoring the high-order byte(s) of the former. Converting a unsigned integer to a signed integer of the same type is only done in administration. This means that the bit-pattern remains unchanged.
- The result of bitwise operations on signed integers are what can be expected on a two's complement machine.
- When either operand is negative, the result of the / operator is the largest integer less than or equal to the algebraic quotient. The sign of the remainder on integer division is the sign of the

enumerator.

- The right-shift of a negative value is negative.

ANS A.6.3.6

- The compiler uses IEEE format for floating-point numbers. High-precision floating-point is used for constant folding.
- Truncation is done to the nearest floating-point number that can be represented.

ANS A.6.3.7

- The type of the sizeof-operator (also known as size_t) is 'unsigned int'.
- Casting an integer to a pointer or vice versa has no effect in bit-pattern when the sizes are equal. Otherwise the value will be truncated or zero-extended (depending on the direction of the conversion and the relative sizes).
- The type of a 'ptrdiff_t' is 'int' on , and 'long' on the 68000 MINIX-PC versions.

ANS A.6.3.8

- Since the front end has only limited control over the registers, it can only make it more likely that variables that are declared as registers also end up in registers. The only things that can possibly be put into registers are plain ints and pointers.

ANS A.6.3.9

- When a member of a union object is accessed using a member of a different type, the resulting value will usually be garbage. The compiler makes no effort to catch these errors.
- The alignment of types under 16-bit is 1 byte for characters and 2 bytes for all other types. Under other Minix versions 'int' and smaller types are aligned to a multiple of their size, bigger scalar types are aligned like 'int'. Arrays have the same alignment as their elements; structs and unions are aligned like their field with the worst alignment.
- A plain 'int' bit-field is taken as a 'signed int'. This means that a field with a size 1 bit-field can only store the values 0 and −1.
- In bit-fields, high-order bits are allocated first.
- An enum has the same size as a plain 'int'.

ANS A.6.3.10

- An access to a volatile object is either a load or a store. Just mentioning a volatile variable is not enough. E.g. the statement 'x;' where x is declared volatile, does not constitute an access. When a volatile object should be read, but its value ignored, 'if (x);' should do the trick.

ANS A.6.3.11

- There is no fixed limit on the number of declarators that may modify an arithmetic, structure or union type, although specifying too many may cause the compiler to run out of memory.

ANS A.6.3.12

- The maximum number of cases in a switch-statement is in the order of 1e9, although the compiler may run out of memory somewhat earlier.

ANS A.6.3.13

- Since both the preprocessor and the compiler assume ASCII-characters, a single character constant in a conditional-inclusion directive matches the same value in the execution character set.
- The preprocessor recognizes *–I...* command-line options. The directories thus specified are searched first. After that, /usr/include is visited.
- Quoted names are first looked for in the directory in which the file which does the include resides.

- The characters in a h- or q- char-sequence are taken to be UNIX paths.
- Neither the front-end nor the preprocessor know any pragmas.
- Since the compiler runs on MINIX __DATE__ and __TIME__ will always be defined.

ANS A.6.3.14

- NULL is defined as ((void *)0). This in order to detect dubious constructions like 'int x = NULL;'.
- The diagnostic printed by 'assert' is as follows:

  Assertion "<expr>" failed, file "<file>", line <line>

  where <expr> is the argument to the assert macro, printed as string. (the <file> and <line> should be clear)
- The sets for character test macros for the C locale are as follows:

Name	**Set**
isalnum	0-9A-Za-z
isalpha	A-Za-z
iscntrl	\000-\037\177
islower	a-z
isupper	A-Z
isprint	\040-\176

  As an addition, there is an *isascii* macro, which tests whether a character is an ASCII character. Characters in the range from \000 to \177 are ASCII characters.
- The behavior of ACK mathematical functions on domain error is as follows:

Name	**Returns**
asin	0.0
acos	0.0
atan2	0.0
fmod	0.0
log	−HUGE_VAL
log10	−HUGE_VAL
pow	0.0
sqrt	0.0

  MINIX-vmd uses the BSD4.4 C library and the Sun FDLIBM C math library instead of the ACK library. See **math**(3) for details about the math functions. The MINIX-vmd libraries offer at least the same functionality as the ACK library.
- Underflow range errors do not cause *errno* to be set.
- The function *fmod* returns 0.0 and sets *errno* to EDOM when the second argument is 0.0.
- The set of signals for the *signal* function is as described by **sigaction**(2).
- A text-stream need not end in a new-line character.
- White space characters before a new-line appear when read in.
- There may be any number of null characters appended to a binary stream.
- The file position indicator of an append mode stream is initially positioned at the beginning of the file.
- A write on a text stream does not cause the associated file to be truncated beyond that point.
- The buffering intended by the standard is fully supported.
- A zero-length file actually exists.
- A file name can consist of any character, except for the '\0' and the '/'.

- A file can be open multiple times.
- When a *remove* is done on an open file, reading and writing behave just as can be expected from a non-removed file. When the associated stream is closed, however, all written data will be lost.
- When a file exists prior to a call to *rename*, it is removed.
- The %p conversion in *fprintf* has the same effect as %#x on and %#lx on the 68000 versions of MINIX-PC.
- The %p conversion in *fscanf* has the same effect as %x on and %lx on the 68000 versions of MINIX-PC.
- A – character that is neither the first nor the last character in the scanlist for %[conversion is taken to be a range indicator. When the first character has a higher ASCII-value than the second, the – will just be put into the scanlist.
- The value of *errno* when *fgetpos* or *ftell* failed is that of *lseek*. This means:

  EBADF − when the stream is not valid
  ESPIPE − when fildes is associated with a pipe
  EINVAL − the resulting file pointer would be negative

- The messages generated by *perror* depend on the value of *errno*. The mapping of errors to strings is done by *strerror*.
- When the requested size is zero, *malloc*, *calloc* and *realloc* return a null-pointer under MINIX-PC. Under MINIX-vmd a unique non-null pointer is returned.
- When *abort* is called, output buffers will be flushed. Temporary files (made with the *tmpfile* function) will have disappeared when SIGABRT is not caught or ignored.
- The *exit* function returns the low-order eight bits of its argument to the environment.
- The predefined environment names are controlled by the user. Setting environment variables is done through the *putenv* function. This function accepts a pointer to char as its argument. To set, for example, the environment variable TERM to a230 one writes

  ```
  static char terminal[] = "TERM=a230";
  putenv(terminal);
  ```

  The argument to *putenv* is stored in an internal table, so malloc'ed strings cannot be freed until another call to *putenv* (which sets the same environment variable) is made. The argument to *putenv* must be writable, which means that officially, the argument cannot be a string constant. The function returns 1 if it fails, 0 otherwise.
- The argument to *system* is passed as argument to */bin/sh –c*.
- The strings returned by *strerror* depend on *errno*. They are listed in **intro**(2). Everything else causes *strerror* to return "unknown error" under MINIX-PC, or the result of sprintf("Error %d", errno) under MINIX-vmd.
- The local time zone is per default GMT. This can be changed through the TZ environment variable, e.g. TZ=EST6. See **TZ**(5).
- The *clock* function returns the number of ticks since process startup.

References

[1] ANS X3.159-1989 *American National Standard for Information Systems - Programming Language C*

THE MINIX MODULA-2 COMPILER

This section describes the implementation-specific features of the MINIX Modula-2 compiler. It is not intended to teach Modula-2 programming. For a description of the Modula-2 language, the reader is referred to [1].

The language implemented

This paragraph discusses the deviations from the Modula-2 language as described in the 'Report on The Programming Language Modula-2', as it appeared in [1], from now on referred to as 'the Report'. Also, the Report sometimes leaves room for interpretation. The section numbers mentioned are the section numbers of the Report.

Syntax (section 2)

The syntax recognized is that of the Report, with some extensions to also recognize the syntax of an earlier definition, given in [2]. Only one compilation unit per file is accepted.

Vocabulary and Representation (section 3)

The input '`10..`' is parsed as two tokens: '`10`' and '`..`'.

The empty string `""` has type

```
ARRAY [0 .. 0] OF CHAR
```

and contains one character: `0C`.

When the text of a comment starts with a '`$`', it may be a pragma. Currently, the following pragmas exist:

```
(*$F      (F stands for Foreign) *)
(*$R[+|-] (Runtime checks, on or off, default on) *)
(*$A[+|-] (Array bound checks, on or off, default off) *)
(*$U      (Allow for underscores within identifiers) *)
```

The Foreign pragma is only meaningful in a `DEFINITION MODULE`, and indicates that this `DEFINITION MODULE` describes an interface to a module written in another language (for instance C or Pascal). Runtime checks that can be disabled are: range checks, `CARDINAL` overflow checks, checks when assigning a `CARDINAL` to an `INTEGER` and vice versa, and checks that `FOR`-loop control-variables are not changed in the body of the loop. Array bound checks can be enabled, because many EM implementations do not implement the array bound checking of the EM array instructions. When enabled, the compiler generates a check before generating an EM array instruction. Even when underscores are enabled, they still may not start an identifier.

Constants of type `LONGINT` are integers with a suffix letter `D` (for instance `1987D`). Constants of type `LONGREAL` have suffix `D` if a scale factor is missing, or have `D` in place of `E` in the scale factor (f.i. `1.0D`, `0.314D1`). This addition was made, because there was no way to indicate long constants, and also because the addition was made in Wirth's newest Modula-2 compiler.

Declarations and scope rules (section 4)

Standard identifiers are predeclared, and valid in all parts of a program. They are called *pervasive*. Unfortunately, the Report does not state how this pervasiveness is accomplished. However, page 87 of [1] states: 'Standard identifiers are automatically imported into all modules'. Our implementation therefore allows redeclarations of standard identifiers within procedures, but not within modules.

Constant expressions (section 5)

Each operand of a constant expression must be a constant: a string, a number, a set, an enumeration literal, a qualifier denoting a constant expression, a type transfer with a constant argument, or one of the standard procedures `ABS`, `CAP`, `CHR`, `LONG`, `MAX`, `MIN`, `ODD`, `ORD`, `SIZE`, `SHORT`, `TSIZE`, or `VAL`, with constant argument(s); `TSIZE` and `SIZE` may also have a variable as argument.

Type declarations (section 6)

1. Basic types (section 6.1)

The type `CHAR` includes the ASCII character set as a subset. Values range from `0C` to `377C`, not from `0C` to `177C`.

2. Enumerations (section 6.2)

The maximum number of enumeration literals in any one enumeration type is `MAX(INTEGER)`.

3. Record types (section 6.5)

The syntax of variant sections in [1] is different from the one in [2]. Our implementation recognizes both, giving a warning for the older one.

4. Set types (section 6.6)

The only limitation imposed by the compiler is that the base type of the set must be a subrange type, an enumeration type, `CHAR`, or `BOOLEAN`. So, the lower bound may be negative. However, if a negative lower bound is used, the compiler gives a warning of the *restricted* class.

The standard type `BITSET` is defined as

```
TYPE BITSET = SET OF [0 .. 8*SIZE(INTEGER)-1];
```

Expressions (section 8)

1. Operators (section 8.2)

1.1. Arithmetic operators (section 8.2.1)

The Report does not specify the priority of the unary operators `+` or `-`: It does not specify whether

```
- 1 + 1
```

means

```
- (1 + 1)
```

or

```
(-1) + 1
```

The MINIX Modula-2 compiler implements the second alternative.

Statements (section 9)

1. Assignments (section 9.1)

The Report does not define the evaluation order in an assignment. Our compiler certainly chooses an evaluation order, but it is explicitly left undefined. Therefore, programs that depend on it may cease to work later.

The types `INTEGER` and `CARDINAL` are assignment-compatible with `LONGINT`, and `REAL` is assignment-compatible with `LONGREAL`.

2. Case statements (section 9.5)

The size of the type of the case-expression must be less than or equal to the word-size.

The Report does not specify what happens if the value of the case-expression does not occur as a label of any case, and there is no `ELSE`-part. In our implementation, this results in a runtime error.

3. For statements (section 9.8)

The Report does not specify the legal types for a control variable. Our implementation allows the basic types (except `REAL`), enumeration types, and subranges. A runtime warning is generated when the value of the control variable is changed by the statement sequence that forms the body of the loop, unless runtime checking is disabled.

4. Return and exit statements (section 9.11)

The Report does not specify which result-types are legal. Our implementation allows any result type.

Procedure declarations (section 10)

Function procedures must exit through a RETURN statement, or a runtime error occurs.

1. Standard procedures (section 10.2)

Our implementation supports `NEW` and `DISPOSE` for backwards compatibility, but issues warnings for their use.

Also, some new standard procedures were added, similar to the new standard procedures in Wirth's newest compiler:

- `LONG` converts an argument of type `INTEGER` or `REAL` to the types `LONGINT` or `LONGREAL`.
- `SHORT` performs the inverse transformation, without range checks.
- `FLOATD` is analogous to `FLOAT`, but yields a result of type `LONGREAL`.
- `TRUNCD` is analogous to `TRUNC`, but yields a result of type `LONGINT`.

System-dependent facilities (section 12)

The type `BYTE` is added to the `SYSTEM` module. It occupies a storage unit of 8 bits. `ARRAY OF BYTE` has a similar effect to `ARRAY OF WORD`, but is safer. In some obscure cases the `ARRAY OF WORD` mechanism does not quite work properly.

The procedure `IOTRANSFER` is not implemented.

Backwards compatibility

Besides recognizing the language as described in [1], the compiler recognizes most of the language described in [2], for backwards compatibility. It warns the user for old-fashioned constructions (constructions that [1] does not allow). If the *–3* option is passed to *m2*, this backwards compatibility feature is disabled.

Compile time errors

The compile time error messages are intended to be self-explanatory, and not listed here. The compiler also sometimes issues warnings, recognizable by a warning-classification between parentheses. There are 3 classifications:

(old-fashioned use)
: These warnings are given on constructions that are not allowed by [1], but are allowed by [2].

(strict)
: These warnings are given on constructions that are supported by the MINIX Modula-2 compiler, but might not be supported by others. Examples: functions returning structured types, SET types of subranges with negative lower bound.

(warning)
: The other warnings, such as warnings about variables that are never assigned, never used, etc.

Runtime errors

The *Traps* module enables the user to install his own runtime error handler. The default one just displays what happened and exits. Basically, a trap handler is just a procedure that takes an INTEGER as parameter. The INTEGER is the trap number. This INTEGER can be one of the EM trap numbers, listed in [3], or one of the numbers listed in the *Traps* definition module.

The following runtime errors may occur:

array bound error
: This error is detected if the *–A* option is given to *m2*.

range bound error
: Range bound errors are always detected, unless runtime checks are disabled.

set bound error

cardinal overflow
: This error is detected, unless runtime checks are disabled.

cardinal underflow
: This error is detected, unless runtime checks are disabled.

divide by 0

divide by 0.0

conversion error
: This error occurs when assigning a negative value of type INTEGER to a variable of type CARDINAL, or when assigning a value of CARDINAL that is > MAX(INTEGER), to a

variable of type INTEGER. It is detected, unless runtime checking is disabled.

heap overflow
: This might happen when ALLOCATE fails.

case error
: This error occurs when non of the cases in a CASE statement are selected, and the CASE statement has no ELSE part.

stack size of process too large
: This is most likely to happen if the reserved space for a coroutine stack is too small. In this case, increase the size of the area given to `NEWPROCESS`. It can also happen if the stack needed for the main process is too large and there are coroutines. In this case, the only fix is to reduce the stack size needed by the main process, f.i. by avoiding local arrays.

too many nested traps + handlers
: This error can only occur when the user has installed his own trap handler. It means that during execution of the trap handler another trap has occurred, and that several times. In some cases, this is an error because of overflow of some internal tables.

no RETURN from function procedure
: This error occurs when a function procedure does not return properly ('falls' through).

illegal instruction
: This error might occur when you use floating point operations on an implementation that does not have floating point.

In addition, some of the library modules may give error messages. The **Traps**-module has a suitable mechanism for this.

The procedure call interface

Parameters are pushed on the stack in reversed order. For VAR parameters, its address is passed, for value parameters its value. The only exception to this rule is with conformant arrays. For conformant arrays, the address is passed, and an array descriptor is passed. The descriptor is an EM array descriptor. It consists of three fields: the lower bound (always 0), upper bound – lower bound, and the size of the elements. The descriptor is pushed first. If the parameter is a value parameter, the called routine must make sure that its value is never changed, for instance by making its own copy of the array.

When the size of the return value of a function procedure is larger than the maximum of `SIZE(LONGREAL)` and twice the pointer-size, the caller reserves this space on the stack, above the parameters. Callee then stores its result there, and returns no other value.

The Modula-2 runtime library

The definition modules of the modules available in the MINIX Modula-2 runtime library reside in the directory */usr/lib/ack/m2*.

References

[1] Niklaus Wirth, *Programming in Modula-2, third, corrected edition*, Springer-Verlag, Berlin (1985)

[2] Niklaus Wirth, *Programming in Modula-2*, Stringer-Verlag, Berlin (1983)

[3] A.S.Tanenbaum, J.W.Stevenson, Hans van Staveren, E.G.Keizer, *Description of a machine architecture for use with block structured languages*, Informatica rapport IR-81, Vrije Universiteit, Amsterdam

THE MINIX PASCAL COMPILER

This section refers to the (1982) BSI standard for Pascal [1]. MINIX Pascal complies with the requirements of level 1 of BS 6192: 1982, with the exceptions as listed in this section.

The standard requires an accompanying document describing the implementation-defined and implementation-dependent features, the reaction on errors and the extensions to standard Pascal. These four items will be treated in the rest of this section.

Implementation-defined features

For each implementation-defined feature mentioned in the BSI standard we give the section number, the quotation from that section and the definition. First we quote the definition of implementation-defined:

> Possibly differing between processors, but defined for any particular processor.

BS 6.1.7: Each string-character shall denote an implementation-defined value of the required char-type.

> All 7-bit ASCII characters except linefeed LF (10) are allowed.

BS 6.4.2.2: The values of type real shall be an implementation-defined subset of the real numbers denoted as specified by 6.1.5 by the signed-real values.

> The set of real values range from a low of −1.7976931348623157e+308 to a high of 1.7976931348623157e+308.

BS 6.4.2.2: The type char shall be the enumeration of a set of implementation-defined characters, some possibly without graphic representations.

> The 7-bit ASCII character set is used, where LF (10) denotes the end-of-line marker on text-files.

BS 6.4.2.2: The ordinal numbers of the character values shall be values of integer-type, that are implementation-defined, and that are determined by mapping the character values on to consecutive non-negative integer values starting at zero.

> The normal ASCII ordering is used: ord('0')=48, ord('A')=65, ord('a')=97, etc.

BS 6.6.5.2: The post-assertions imply corresponding activities on the external entities, if any, to which the file-variables are bound. These activities, and the point at which they are actually performed, shall be implementation-defined.

> The reading and writing writing of objects on files is buffered. This means that when a program terminates abnormally, I/O may be unfinished. Terminal I/O is unbuffered. Files are closed whenever they are rewritten or reset, or on program termination.

BS 6.7.2.2: The predefined constant *maxint* shall be of integer-type and shall denote an implementation-defined value, that satisfies the following conditions:

(a) All integral values in the closed interval from −*maxint* to +*maxint* shall be values of the integer-type.

(b) Any monadic operation performed on an integer value in this interval shall be correctly performed according to the mathematical rules for integer arithmetic.

(c) Any dyadic integer operation on two integer values in this same interval shall be correctly performed according to the mathematical rules for integer arithmetic, provided that the result is also in this interval.

(d) Any relational operation on two integer values in this same interval shall be correctly performed according to the mathematical rules for integer arithmetic.

> The representation of integers under 16-bit or under 68000 MINIX-PC is a 16-bit word using two's complement arithmetic. The integers range from −32768 to +32767. Under 32-bit a 32-bit integer is used ranging from −2147483648 to +2147483647.

BS 6.7.2.2: The result of the real arithmetic operators and functions shall be approximations to the corresponding mathematical results. The accuracy of this approximation shall be implementation-defined

> The default size of reals is 8 bytes, the accuracy is 11 bits for the exponent, and 53 bits for the mantissa. This gives an accuracy of about 16 digits. and exponents ranging from −307 to +307.

BS 6.9.3.1: The default TotalWidth values for integer, Boolean and real types shall be implementation-defined.

The defaults are:

integer	6	(16-bit)
integer	11	(32-bit)
Boolean	5	
real	14	

BS 6.9.3.4.1: ExpDigits, the number of digits written in an exponent part of a real, shall be implementation-defined.

ExpDigits is defined as 3.

BS 6.9.3.4.1: The character written as part of the representation of a real to indicate the beginning of the exponent part shall be implementation-defined, either 'E' or 'e'.

The exponent part starts with 'e'.

BS 6.9.3.5: The case of the characters written as representation of the Boolean values shall be implementation-defined.

The representations of true and false are 'true' and 'false'.

BS 6.9.5: The effect caused by the standard procedure page on a text file shall be implementation-defined.

The ASCII character form feed FF (12) is written.

BS 6.10: The binding of the variables denoted by the program-parameters to entities external to the program shall be implementation-defined if the variable is of a file-type.

The program parameters must be files and all, except input and output, must be declared as such in the program block.

The program parameters input and output, if specified, will correspond with the UNIX streams 'standard input' and 'standard output'.

The other program parameters will be mapped to the argument strings provided by the caller of this program. The argument strings are supposed to be path names of the files to be opened or created. The order of the program parameters determines the mapping: the first parameter is mapped onto the first argument string, etc. Note that input and output are ignored in this mapping.

The mapping is recalculated each time a program parameter is opened for reading or writing by a call to the standard procedures reset or rewrite. This gives the programmer the opportunity to manipulate the list of string arguments using the external procedures argc, argv and argshift available in the Pascal library.

BS 6.10: The effect of an explicit use of reset or rewrite on the standard text files input or output shall be implementation-defined.

The procedures reset and rewrite are no-ops if applied to input or output.

Implementation-dependent features

For each implementation-dependent feature mentioned in the BSI standard, we give the section number, the quotation from that section and the way this feature is treated by the MINIX Pascal system. First we quote the definition of 'implementation-dependent':

Possibly differing between processors and not necessarily defined for any particular processor.

BS 6.7.2.1: The order of evaluation of the operands of a dyadic operator shall be implementation-dependent.

Operands are always evaluated, so the program part

```
if (p<>nil) and (p^.value<>0) then
```

is probably incorrect.

The left-hand operand of a dyadic operator is almost always evaluated before the right-hand side. Some peculiar evaluations exist for the following cases:

1. The modulo operation is performed by a library routine to check for negative values of the right operand.

2. The expression

   set1 <= set2

   where set1 and set2 are compatible set types is evaluated in the following steps:

   - evaluate set2;
   - evaluate set1;
   - compute set2+set1;
   - test set2 and set2+set1 for equality.

3. The expression

   set1 >= set2

   where set1 and set2 are compatible set types is evaluated in the following steps:

   - evaluate set1;
   - evaluate set2;
   - compute set1+set2;
   - test set1 and set1+set2 for equality.

BS 6.7.3: The order of evaluation, accessing and binding of the actual-parameters for functions shall be implementation-dependent.

The order of evaluation is from right to left.

BS 6.8.2.2: The decision as to the order of accessing the variable and evaluating the expression in an assignment-statement, shall be implementation-dependent.

The expression is evaluated first.

BS 6.8.2.3: The order of evaluation and binding of the actual-parameters for procedures shall be implementation-dependent.

The same as for functions.

BS 6.9.5: The effect of inspecting a text file to which the page procedure was applied during generation is implementation-dependent.

The formfeed character written by page is treated like a normal character, with ordinal value 12.

BS 6.10: The binding of the variables denoted by the program-parameters to entities external to the program shall be implementation-dependent unless the variable is of a file-type.

Only variables of a file-type are allowed as program parameters.

Error handling

There are three classes of errors to be distinguished. In the first class are the error messages generated by the compiler. The second class consists of the occasional errors generated by the other programs involved in the compilation process. Errors of the third class are the errors as defined in the standard by:

> An error is a violation by a program of the requirements of this standard that a processor is permitted to leave undetected.

Compiler errors

Error are written on the standard error output. Each line has the form:

<file>, line <number>: <description>

Every time the compiler detects an error that does not have influence on the code produced by the compiler or on the syntax decisions, a warning messages is given. If only warnings are generated, compilation proceeds and probably results in a correctly compiled program.

Sometimes the compiler produces several errors for the same line. They are only shown up to a maximum of 5 errors per line. Warning are also shown up to a maximum of 5 per line.

Extensive treatment of these errors is outside the scope of this manual.

Runtime errors

Errors detected at run time cause an error message to be generated on the diagnostic output stream (UNIX file descriptor 2). The message consists of the name of the program followed by a message describing the error, possibly followed by the source line number. Unless the *–n* option is turned on, the compiler generates code to keep track of which source line causes which instructions to be generated.

For each error mentioned in the standard we give the section number, the quotation from that section and the way it is processed by the Pascal-compiler or runtime system.

For detected errors the corresponding message and trap number are given. Trap numbers are useful for exception-handling routines. Normally, each error causes the program to terminate. By using exception-handling routines one can ignore errors or perform alternate actions. Only some of the errors can be ignored by restarting the failing instruction. These errors are marked as non-fatal, all others as fatal. A list of errors with trap number between 0 and 63 (EM errors) can be found in [2]. Errors with trap number between 64 and 127 (Pascal errors) are listed below.

BS 6.4.6: It shall be an error if a value of type T2 must be assignment-compatible with type T1, while T1 and T2 are compatible ordinal-types and the value of type T2 is not in the closed interval specified by T1.

> The compiler distinguishes between array-index expressions and the other places where assignment-compatibility is required.

Array subscripting errors are only detected when the 'A' option is used. In the other cases, a range bound error occurs when the value of type T2 is not in the closed interval specified by T1, unless range checks are disabled.

BS 6.4.6: It shall be an error if a value of type T2 must be assignment-compatible with type T1, while T1 and T2 are compatible set-types and any member of the value of type T2 is not in the closed interval specified by the base-type of the type T1.

> This error is not detected.

BS 6.5.3.3: It shall be an error if a component of a variant-part of a variant, where the selector of the variant-part is not a field, is accessed unless the variant is active for the entirety of each reference and access to each component of the variant.

> This error is not detected.

BS 6.5.4: It shall be an error if the pointer-variable of an identified-variable either denotes a nil-value or is undefined.

> This error is not detected.

BS 6.5.4: It shall be an error to remove the identifying-value of an identified variable from its pointer-type when a reference to the variable exists.

> When the identified variable is an element of the record-variable-list of a with-statement, a warning is given at compile-time. Otherwise, this error is not detected.

BS 6.5.5: It shall be an error to alter the value of a file-variable f when a reference to the buffer-variable f^ exists.

> When f is altered when it is an element of the record-variable-list of a with-statement, a warning is given. When a buffer-variable is used as a variable-parameter, an error is given. This is done at compile-time.

BS 6.6.5.2: It shall be an error if the stated pre-assertion does not hold immediately prior to any use of the file handling procedures rewrite, put, reset and get.

For each of these four operations the pre-assertions can be reformulated as:

rewrite(f):	no pre-assertion.
put(f):	f is opened for writing and fˆ is not undefined.
reset(f):	f exists.
get(f):	f is opened for reading and eof(f) is false.

The following errors are detected for these operations:

rewrite(f):

more args expected, trap 64, fatal:
f is a program-parameter and the corresponding file name is not supplied by the caller of the program.

rewrite error, trap 101, fatal:
the caller of the program lacks the necessary access rights to create the file in the file system or operating system problems like table overflow prevent creation of the file.

put(f):

file not yet open, trap 72, fatal:
reset or rewrite are never applied to the file. The checks performed by the run time system are not foolproof.

not writable, trap 96, fatal:
f is opened for reading.

write error, trap 104, fatal:
probably caused by file system problems. For instance, the file storage is exhausted. Because I/O is buffered to improve performance, it might happen that this error occurs if the file is closed. Files are closed whenever they are rewritten or reset, or on program termination.

reset(f):

more args expected, trap 64, fatal:
same as for rewrite(f).

reset error, trap 100, fatal:
f does not exist, or the caller has insufficient access rights, or operating system tables are exhausted.

get(f):

file not yet open, trap 72, fatal:
as for put(f).

not readable, trap 97, fatal:
f is opened for writing.

end of file, trap 98, fatal:
eof(f) is true just before the call to get(f).

read error, trap 103, fatal:
unlikely to happen. Probably caused by hardware problems or by errors elsewhere in your program that destroyed the file information maintained by the run time system.

truncated, trap 99, fatal:
the file is not properly formed by an integer number of file elements. For instance, the size of a file of integer is odd.

non-ASCII char read, trap 106, non-fatal:
the character value of the next character-type file element is out of range (0..127). Only for text files.

BS 6.6.5.3: It shall be an error if a variant of a variant-part within the new variable becomes active and a different variant of the variant-part is one of the specified variants.

This error is not detected.

BS 6.6.5.3: It shall be an error to use dispose(q) if the identifying variable has been allocated using the form new(p,c1,...,cn).

This error is not detected. However, this error can cause more memory to be freed then was allocated. Dispose causes a fatal trap 73 when memory already on the free list is freed again.

BS 6.6.5.3: It shall be an error to use dispose(q,k1,...,km) if the identifying variable has been allocated using the form new(p,c1,...,cn) and m is not equal to n.

This error is not detected. However, this error can cause more memory to be freed then was allocated. Dispose causes a fatal trap 73 when memory already on the free list is freed again.

BS 6.6.5.3: It shall be an error if the variants of a variable to be disposed are different from those specified by the case-constants to dispose.

This error is not detected.

BS 6.6.5.3: It shall be an error if the value of the pointer parameter of dispose has nil-value or is undefined.

This error is detected for nil-value (dispose error, trap 73, fatal).

BS 6.6.5.3: It shall be an error if a variable created using the second form of new is accessed by the identified variable of the variable-access of a factor, of an assignment-statement, or of an actual-parameter.

This error is not detected.

BS 6.6.6.2: It shall be an error if the value of sqr(x) does not exist.

This error is detected for real-type arguments (real overflow, trap 4, non-fatal).

BS 6.6.6.2: It shall be an error if x in ln(x) is smaller than or equal to 0.

This error is detected (error in ln, trap 66, non-fatal)

BS 6.6.6.2: It shall be an error if x in sqrt(x) is smaller than 0.

This error is detected (error in sqrt, trap 67, non-fatal)

In addition to these errors, overflow in the expression exp(x) is detected (error in exp, trap 65, non-fatal; real overflow, trap 4, non-fatal)

BS 6.6.6.3: It shall be an error if the integer value of trunc(x) does not exist.

This error is detected (conversion error, trap 10, non-fatal).

BS 6.6.6.3: It shall be an error if the integer value of round(x) does not exist.

This error is detected (conversion error, trap 10, non-fatal).

BS 6.6.6.4: It shall be an error if the integer value of ord(x) does not exist.

This error can not occur, because the compiler will not allow such ordinal types.

BS 6.6.6.4: It shall be an error if the character value of chr(x) does not exist.

This error is detected (range bound error, trap 1, non-fatal).

BS 6.6.6.4: It shall be an error if the value of succ(x) does not exist.

Same comments as for chr(x).

BS 6.6.6.4: It shall be an error if the value of pred(x) does not exist.

Same comments as for chr(x).

BS 6.6.6.5: It shall be an error if f in eof(f) is undefined.

This error is detected (file not yet open, trap 72, fatal).

BS 6.6.6.5: It shall be an error if f in eoln(f) is undefined, or if eof(f) is true at that time.

The following errors may occur:

file not yet open, trap 72, fatal;
not readable, trap 97, fatal;
end of file, trap 98, fatal.

BS 6.7.1: It shall be an error if a variable-access used as an operand in an expression is undefined at the time of its use.

The compiler performs some limited checks to see if identifiers are used before they are set. Since it can not always be sure (one could, for instance, jump out of a loop), only a warning is generated. When an expression contains a function-call, an error occurs if the function is not assigned at run-time.

BS 6.7.2.2: A term of the form x/y shall be an error if y is zero.

This error is detected (divide by 0.0, trap 7, non-fatal).

BS 6.7.2.2: It shall be an error if j is zero in 'i div j'.

This error is detected (divide by 0, trap 6, non-fatal).

BS 6.7.2.2: It shall be an error if j is zero or negative in i MOD j.

This error is detected (only positive j in 'i mod j', trap 71, non-fatal).

BS 6.7.2.2: It shall be an error if the result of any operation on integer operands is not performed according to the mathematical rules for integer arithmetic.

This implementation does not detect integer overflow.

BS 6.8.3.5: It shall be an error if none of the case-constants is equal to the value of the case-index upon entry to the case-statement.

This error is detected (case error, trap 20, fatal).

BS 6.9.1: It shall be an error if the sequence of characters read looking for an integer does not form a signed-integer as specified in 6.1.5.

This error is detected (digit expected, trap 105, non-fatal).

BS 6.9.1: It shall be an error if the sequence of characters read looking for a real does not form a signed-number as specified in 6.1.5.

This error is detected (digit expected, trap 105, non-fatal).

BS 6.9.1: When read is applied to f, it shall be an error if the buffer-variable f^ is undefined or the pre-assertions for get do not hold.

This error is detected (see get(f)).

BS 6.9.3: When write is applied to a text file f, it shall be an error if f is undefined or f is opened for reading.

This error is detected (see put(f)). Furthermore, this error is also detected when f is not a text file.

BS 6.9.3.1: The values of TotalWidth or FracDigits shall be greater than or equal to one; it shall be an error if either value is less then one.

When either value is less than zero, an error (illegal field width, trap 75, non-fatal) occurs. Zero values are allowed, in order to maintain some compatibility with the old MINIX Pascal compiler.

BS 6.9.5: It shall be an error if the pre-assertion required for writeln(f) doe not hold prior to the invocation of page(f);

This error is detected (see put(f)).

Extensions to the standard

1. External routines

Except for the required directive 'forward' the MINIX Pascal compiler recognizes the directive 'extern'. This directive tells the compiler that the procedure block of this procedure will not be present in the current program. The code for the body of this procedure must be included at a later stage of the compilation process.

This feature allows one to build libraries containing often used routines. These routines do not have to be included in all the programs using them. Maintenance is much simpler if there is only one library module to be changed instead of many Pascal programs.

Another advantage is that these library modules may be written in a different language, for instance C.

The use of external routines, however, is dangerous. The compiler normally checks for the correct number and type of parameters when a procedure is called and for the result type of functions. If an external routine is called these checks are not sufficient, because the compiler can not check whether the procedure heading of the external routine as given in the Pascal program matches the actual routine implementation. It should be the loader's task to check this. However, the current loaders are not that smart.

For those who wish the use the interface between C and Pascal we give an incomplete list of corresponding formal parameters in C and Pascal.

Pascal	**C**
a:integer	int a
a:char	int a
a:boolean	int a
a:real	double a
a:^type	type *a
var a:type	type *a

procedure a(pars)	struct {
	  void (*a)() ;
	  char *static_link ;
	}

function a(pars):type	struct {
	  type (*a)() ;
	  char *static_link ;
	}

The Pascal runtime system uses the following algorithm when calling function/procedures passed as parameters.

```
if (static_link) {
        (*a)(static_link, pars);
} else {
        (*a)(pars);
}
```

2. Separate compilation.

The compiler is able to (separately) compile a collection of declarations, procedures and functions to form a library. The library may be linked with the main program, compiled later. The syntax of these modules is

```
module = [constant-definition-part]
         [type-definition-part]
         [var-declaration-part]
         [procedure-and-function-declaration-part]
```

The compiler accepts a program or a module:

```
unit = program | module
```

All variables declared outside a module must be imported by parameters, even the files input and output. Access to a variable declared in a module is only possible using the procedures and functions declared in that same module. By giving the correct procedure/function heading followed by the directive 'extern' you may use procedures and functions declared in other units.

3. Assertions.

When the s-option is off, MINIX Pascal compiler recognizes an additional statement, the assertion. Assertions can be used as an aid in debugging and documentation. The syntax is:

```
assertion = 'assert' Boolean-expression
```

An assertion is a simple-statement, so

```
simple-statement = [assignment-statement |
                    procedure-statement |
                    goto-statement |
                    assertion
                   ]
```

An assertion causes an error if the Boolean-expression is false. That is its only purpose. It does not change any of the variables, at least it should not. Therefore, do not use functions with side-effects in the Boolean-expression. If the a-option is turned on, then assertions are skipped by the compiler. 'assert' is not a word-symbol (keyword) and may be used as identifier. However, assignment to a variable and calling of a procedure with that name will be impossible. If the s-option is turned on, the compiler will not know a thing about assertions, so using assertions will then give a parse error.

4. Additional procedures.

Three additional standard procedures are available:

halt: a call of this procedure is equivalent to jumping to the end of your program. It is always the last statement executed. The exit status of the program may be supplied as optional argument. If not, it will be zero.

release:

mark: for most applications it is sufficient to use the heap as second stack. Mark and release are suited for this type of use, more suited than dispose. mark(p), with p of type pointer, stores the current value of the heap pointer in p. release(p), with p initialized by a call of mark(p), restores the heap pointer to its old value. All the heap objects, created by calls of new between the call of mark and the call of release, are removed and the space they used can be reallocated. Never use mark and release together with dispose!

5. UNIX interfacing.

If the c-option is turned on, then some special features are available to simplify an interface with the UNIX environment. First of all, the compiler allows you to use a different type of string constants. These string constants are delimited by double quotes ('"'). To put a double quote into these strings, you must repeat the double quote, like the single quote in normal string constants. These special string constants are terminated by a zero byte (chr(0)). The type of these constants is a pointer to a packed array of characters, with lower bound 1 and unknown upper bound.
Secondly, the compiler predefines a new type identifier 'string' denoting this just described string type.

The only thing you can do with these features is declaration of constants and variables of type 'string'. String objects may not be allocated on the heap and string pointers may not be de-referenced. Still these strings are very useful in combination with external routines. The procedure write is extended to print these zero-terminated strings correctly.

6. Double length (32 bit) integers.

If the d-option is turned on, then the additional type 'long' is known to the compiler. Long variables have integer values in the range −2147483648 .. +2147483647. Long constants can not be declared. Longs can not be used as control-variables. It is not allowed to form subranges of type long. All operations allowed on integers are also allowed on longs and are indicated by the same operators: '+',

'-', '*', '/', 'div', 'mod'. The procedures read and write have been extended to handle long arguments correctly. It is possible to read longs from a file of integers and vice-versa, but only if longs and integers have the same size. The default width for longs is 11. The standard procedures 'abs' and 'sqr' have been extended to work on long arguments. Conversion from integer to long, long to real, real to long and long to integer are automatic, like the conversion from integer to real. These conversions may cause a

conversion error, trap 10, non-fatal

7. Underscore as letter.

The character '_' may be used in forming identifiers, if the u- or U-option is turned on. It is forbidden to start identifiers with underscores, since this may cause name-clashes with run-time routines.

8. Zero field width in write.

Zero TotalWidth arguments are allowed. In this case, no characters are written for character, string or Boolean type arguments. A zero FracDigits argument for fixed-point representation of reals causes the fraction and the character '.' to be suppressed.

9. Pre-processing.

If the very first character of a file containing a Pascal program is the sharp ('#', ASCII 23(hex)) the file is preprocessed in the same way as C programs. Lines beginning with a '#' are taken as preprocessor command lines and not fed to the Pascal compiler proper. C style comments, /*.....*/, are removed by the C preprocessor, thus C comments inside Pascal programs are also removed when they are fed through the preprocessor.

Deviations from the standard

MINIX Pascal deviates from the standard in the following ways:

1. Standard procedures and functions are not allowed as parameters in MINIX Pascal. You can obtain the same result with negligible loss of performance by declaring some user routines like:

```
function sine(x:real):real;
begin
    sine:=sin(x)
end;
```

2. The standard procedures read, readln, write and writeln are implemented as word-symbols, and can therefore not be redeclared.

Compiler options

Some options of the compiler may be controlled by using '{$....}'. Each option consists of a lower case letter followed by +, – or an unsigned number. Options are separated by commas. The following options exist:

a+/– This option switches assertions on and off. If this option is on, then code is included to test these assertions at run time. Default +.

c+/– This option, if on, allows you to use C-type string constants surrounded by double quotes. Moreover, a new type identifier 'string' is predefined. Default –.

d+/– This option, if on, allows you to use variables of type 'long'. Default –.

i<num>
With this flag the setsize for a set of integers can be manipulated. The number must be the number of bits per set. The default value is 16.

l+/– If + then code is inserted to keep track of the source line number. When this flag is switched on and off, an incorrect line number may appear if the error occurs in a part of your program for which this flag is off. Default +.

r+/– If + then code is inserted to check subrange variables against lower and upper subrange limits. Default +.

s+/– If + then the compiler will hunt for places in your program where non-standard features are used, and for each place found it will generate a warning. Default –.

t+/– If + then each time a procedure is entered, the routine 'procentry' is called, and each time a procedure exits, the procedure 'procexit' is called. Both 'procentry' and 'procexit' have a 'string' as parameter. This means that when a user specifies his or her own procedures, the c-option must be used. Default procedures are present in the run time library. Default –.

u+/– If + then the character '_' is treated like a letter, so that it may be used in identifiers. Procedure and function identifiers are not allowed to start with an underscore because they may collide with library routine names. Default –.

Some of these flags (c, d, i, s, u, C and U) are only effective when they appear before the 'program' symbol. The others may be switched on and off.

A very powerful debugging tool is the knowledge that inaccessible statements and useless tests are removed by the optimizer. For instance, a statement like:

```
if debug then
    writeln('initialization done');
```

is completely removed by the optimizer if debug is a constant with value false. The first line is removed if debug is a constant with value true. Of course, if debug is a variable nothing can be removed.

Library routines

The following library of external routines for Pascal programs is available:

```
const     bufsize = ?;
type      br1 =  1..bufsize;
          br2 =  0..bufsize;
          br3 = -1..bufsize;
          ok = -1..0;
          buf = packed array[br1] of char;
          alfa = packed array[1..8] of char;
          string = ^packed array[1..?] of char;
          filetype = file of ?;
          long = ?;

{all routines must be declared extern}

function  argc:integer;
function  argv(i:integer):string;
function  environ(i:integer):string;
procedure argshift;

procedure buff(var f:filetype);
procedure nobuff(var f:filetype);
procedure notext(var f:text);
procedure diag(var f:text);
procedure pcreat(var f:text; s:string);
procedure popen(var f:text; s:string);
procedure pclose(var f:filetype);

procedure trap(err:integer);
procedure encaps(procedure p; procedure q(n:integer));

function  perrno:integer;
function  uread(fd:integer; var b:buf; len:br1):br3;
function  uwrite(fd:integer; var b:buf; len:br1):br3;

function  strbuf(var b:buf):string;
function  strtobuf(s:string; var b:buf; len:br1):br2;
function  strlen(s:string):integer;
```

```
function  strfetch(s:string; i:integer):char;
procedure strstore(s:string; i:integer; c:char);
function  clock:integer;
```

This library contains some often used external routines for Pascal programs. The routines can be divided into several categories:

Argument control:

argc — Gives the number of arguments provided when the program is called.

argv — Selects the specified argument from the argument list and returns a pointer to it. This pointer is nil if the index is out of bounds (<0 or >=argc).

environ — Returns a pointer to the i-th environment string (i>=0). Returns nil if i is beyond the end of the environment list (UNIX version 7).

argshift — Effectively deletes the first argument from the argument list. Its function is equivalent to *shift* in the UNIX shell: argv[2] becomes argv[1], argv[3] becomes argv[2], etc. It is a useful procedure to skip optional flag arguments. Note that the matching of arguments and files is done at the time a file is opened by a call to reset or rewrite.

Additional file handling routines:

buff — Turn on buffering of a file. Not very useful, because all files are buffered except standard output to a terminal and diagnostic output. Input files are always buffered.

nobuff — Turn off buffering of an output file. It causes the current contents of the buffer to be flushed.

notext — Only useful for input files. End of line characters are not replaced by a space and character codes out of the ASCII range (0..127) do not cause an error message.

diag — Initialize a file for output on the diagnostic output stream (fd=2). Output is not buffered.

pcreat — The same as rewrite(f), except that you must provide the file name yourself. The name must be zero terminated. Only text files are allowed.

popen — The same as reset(f), except that you must provide the file name yourself. The name must be zero terminated. Only text files are allowed.

pclose — Gives you the opportunity to close files hidden in records or arrays. All other files are closed automatically.

String handling:

strbuf — Type conversion from character array to string. It is your own responsibility that the string is zero terminated.

strtobuf — Copy string into buffer until the string terminating zero byte is found or until the buffer if full, whatever comes first. The zero byte is also copied. The number of copied characters, excluding the zero byte, is returned. So if the result is equal to the buffer length, then the end of buffer is reached before the end of string.

strlen — Returns the string length excluding the terminating zero byte.

strfetch — Fetches the i-th character from a string. There is no check against the string length.

strstore — Stores a character in a string. There is no check against string length, so this is a dangerous procedure.

Trap handling:

These routines allow you to handle almost all the possible error situations yourself. You may define your own trap handler, replacing the default handler that produces an error message and quits. You may also generate traps yourself.

trap Trap generates the trap passed as argument (0..252). The trap numbers 128..252 may be used freely. The others are reserved.

encaps Encapsulate the execution of *p* with the trap handler *q*. Encaps replaces the previous trap handler by *q*, calls *p* and restores the previous handler when *p* returns. If, during the execution of *p*, a trap occurs, then *q* is called with the trap number as parameter. For the duration of *q* the previous trap handler is restored, so that you may handle only some of the errors in *q*. All the other errors must then be raised again by a call to *trap*.
Encapsulations may be nested: you may encapsulate a procedure while executing an encapsulated routine.
Jumping out of an encapsulated procedure (non-local goto) is dangerous, because the previous trap handler must be restored. Therefore, you may only jump out of procedure *p* from inside *q* and you may only jump out of one level of encapsulation. If you want to exit several levels of encapsulation, use traps. See pc_prlib(7) for lists of trap numbers for EM machine errors and Pascal run time system errors. Note that *p* may not have parameters.

UNIX system calls:

uread Equal to the read system call. Its normal name is blocked by the standard Pascal routine read.

uwrite As above but for write(2).

perrno Because external data references are not possible in Pascal, this routine returns the global variable *errno*, indicating the result of the last system call.

Miscellaneous:

clock Return the number of ticks of user and system time consumed by the program.

The following program presents an example of how these routines can be used. This program is equivalent to the UNIX command cat(1).

```
{$c+}
program cat(input,inp,output);
var       inp:text;
          s:string;

function argc:integer; extern;
function argv(i:integer):string; extern;
procedure argshift; extern;
function strlen(s:string):integer; extern;
function strfetch(s:string; i:integer):char; extern;

procedure copy(var fi:text);
var c:char;
begin reset(fi);
  while not eof(fi) do
  begin
    while not eoln(fi) do
    begin
      read(fi,c);
      write(c)
    end;
    readln(fi);
    writeln
  end
end;

begin  {main}
  if argc = 1 then
```

```
            copy(input)
  else
    repeat
      s := argv(1);
      if (strlen(s) = 1) and (strfetch(s,1) = '-')
      then copy(input)
      else copy(inp);
      argshift;
    until argc <= 1;
end.
```

Another example gives some idea of the way to manage trap handling:

```
program bigreal(output);
const EFOVFL=4;
var trapped:boolean;

procedure encaps(procedure p; procedure q(n:integer)); extern;
procedure trap(n:integer); extern;

procedure traphandler(n:integer);
begin if n=EFOVFL then trapped:=true else trap(n) end;

procedure work;
var i,j:real;
begin trapped:=false; i:=1;
  while not trapped do
    begin j:=i; i:=i*2 end;
  writeln('bigreal = ',j);
end;

begin
  encaps(work,traphandler);
end.
```

Two routines may cause fatal error messages to be generated. These are:

pcreat Rewrite error (trap 77) if the file cannot be created.

popen Reset error (trap 76) if the file cannot be opened for reading

References

[1] BSI standard BS 6192: 1982 (ISO 7185).

[2] A.S.Tanenbaum, J.W.Stevenson, Hans van Staveren, E.G.Keizer, "Description of a machine architecture for use with block structured languages", Informatica rapport IR-81.