Pascal-VU reference manual

by

Johan W. Stevenson

Wiskundig Seminarium,
Vrije Universteit,
De Boelelaan 1081,
Amsterdam.

1. Introduction

This document refers to the (March 1980) ISO standard proposal for Pascal [1]. Pascal-VU complies with the requirements of this proposal almost completely. 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 document, each in a separate chapter. The other chapters describe the deviations from the standard and the list of options recognized by the compiler.

The Pascal-VU compiler produces code for an EM-1 machine as defined in [2]. It is up to the implementor of the EM-1 machine to decide whether errors like integer overflow, undefined operand and range bound error are recognized or not. For these errors the reaction of all known implementations is given.

There does not (yet) exist a hardware EM-1 machine. Therefore, EM-1 programs must be interpreted, or translated into instructions for a target machine. The following implementations currently exist:

1. an interpreter running on a PDP-11. Normally the interpreter performs some tests to detect undefined integers, integer overflow, range errors, etc. However, an option of the interpreter is to skip these tests. Another option is to perform some extra tests to check for instance the number of actual parameter words against the number expected by the called procedure. We will refer to these modes of operation as 'test all', 'test on' and 'test off'.

2. a translator into PDP-11 instructions.

ISO 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-bits ASCII character set is used, where LF (10) denotes the end-of-line marker on text-files.

ISO 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.

ISO 6.4.3.4: The largest and smallest values of integer-type permitted as numbers of a value of a set-type shall be implementation-defined.

> The smallest value is 0. The largest value is default 15, but can be changed by using the i-option of the compiler up to a maximum of 32767. The compiler allocates as many bits for set-type variables as are necessary to store all possible values of the host-type of the base-type of the set, rounded up to the nearest multiple of 16. If 8 bits are sufficient then only 8 bits are used if part of a packed structure. Thus, the variable s, declared by

```
      var s: set of '0'..'9';
```

> will contain 128 bits, not 10 or 16. These 128 bits are stored in 16 bytes, both for packed and unpacked sets. If the host-type of the base-type is integer, then the number of bits depends on the i-option. The programmer may specify how many bits to allocate for these sets. The default is 16, the maximum is 32767. The effective number of bits is rounded up to the next multiple of 16, or up to 8 if the number of bits is less than or equal to 8. Note that the use of set-constructors for sets with more than 256 elements is far less efficient than for smaller sets.

ISO 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 in 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.

2. Implementation-defined features

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

> Those parts of the language which may differ between processors, but which will be defined for any particular processor.

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

> All 7-bits ASCII characters except linefeed LF (10) are allowed. Note that an apostrophe ' must be doubled within a string.

ISO 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.

> The format of reals is not defined in EM-1. It is only defined that a real number occupies 2 words (32 bits) of storage, but this might change to 4 words in the future. The compiler can be instructed, by the f-option, to use a different size for real values. For each implementation of EM-1 the following constants must be defined:
>
> epbase: the base for the exponent part
> epprec: the precision of the fraction
> epemin: the minimum exponent
> epemax: the maximum exponent
>
> These constants must be chosen so that zero and all numbers with exponent e in the range
>
> epemin <= e <= epemax
>
> and fraction-parts of the form
>
> f = + f918.b8-19 + ... + f9epprec8.b8-epprec

9 where

> f9i8 = 0,...,epbase-1 and f918 <> 0
>
> are possible values for reals. All other values of type real are considered illegal. (See [3] for more information about these constants).
> For the known EM-1 implementations these constants are:
>
> 1. epbase = 2
>   epprec = 24
>   epemin = -127
>   epemax = +127
>
> 2. ditto

The representation of integers in EM-1 is a 16-bit word using two's complement arithmetic. Thus always:

```
maxint = 32767
```

Because the number -32768 may be used to indicate 'undefined', the range of available integers depends on the EM-1 implementation:

1. test on: -32767..+32767.
   test off: -32768..+32767.

2. -32768..+32767.

ISO 6.9.4.2: The default TotalWidth values for integer, Boolean and real types shall be implementation-defined.

The defaults are:

integer	6
Boolean	5
real	13

ISO 6.9.4.5.1: ExpDigits, the number of digits written in an exponent part of a real, shall be implementation-defined.

ExpDigits is defined as

```
ceil(log10(log10(2 ** epemax)))
```

For the current implementations this evaluates to 2.

ISO 6.9.4.5.1: 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'.

ISO 6.9.4.6: 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'.

ISO 6.9.6: 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.

ISO 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 libpc [7].

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

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

3. Implementation-dependent features

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

> Those parts of the language which may differ between processors, and for which there need not be a definition for a particular processor.

ISO 5.1.1: The method for reporting errors or warnings shall be implementation-dependent.

The error handling is treated in a following chapter.

ISO 6.1.4: Other implementation-dependent directives may be defined.

Except for the required directive 'forward' the Pascal-VU compiler recognizes only one 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 or the EM-1 assembly language. This is useful if you want to use some specific EM-1 instructions not generated by the Pascal compiler. Examples are the system call routines and some floating point conversion routines. Another motive could be the optimization of some time-critical program parts.

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. Another solution is to check at run time, at least the number of words for parameters. Some EM-1 implementations check this:

i. test all: the number of words passed as parameters is checked, but this will not catch all faulty cases.
test on: not checked.

2. not checked.

ISO 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

This is the only case where the right-hand side is computed first.

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

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

The order of evaluation and binding is from left to right.

ISO 6.8.2.2: If access to the variable in an assignment-statement involves the indexing of an array and/or a reference to a field within a variant of a record and/or the de-referencing of a pointer-variable and/or a reference to a buffer-variable, the decision whether these actions precede or follow the evaluation of the expression shall be implementation-dependent.

The expression is evaluated first.

ISO 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.

ISO 6.9.6: 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.

ISO 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.

4. 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 such that detection normally requires execution of the program.

4.1. Compiler errors

The error messages (and the listing) are not generated by the compiler itself. The compiler only detects errors and writes the errors in condensed form on an intermediate file. Each error in condensed form contains:

- an optional error message parameter (identifier or number).
- an error number
- a line number
- a column number

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.

The intermediate error file is read by the interface program pc [4], that produces the error messages. It uses an other file, the error message file, indexed by the error number, to find an error script line. Whenever this error script line contains the character '%', the error messages parameter is substituted. For negative error numbers the message constructed is prepended with 'Warning: '.

Sometimes the compiler produces several errors for the same file position (line number, column number). Only the first of these messages is given, because the others are probably directly caused by the first one. .If the first one is a warning while one of its successors for that position is a fatal message, then the warning is promoted to a fatal one. However, parameterized messages are always given.

The error messages and listing come in three flavors, selected by flags given to pc [4]:

default:no listing, one line per error giving the file name of the Pascal source file, the line number and the error messages.

-e: for each erroneous line a listing of the line and its predecessor. The next line contains one or more characters '^' pointing to the places where an error is detected. For each error on that line a message follows.

-E: same as for '-e', except that all source lines are listed, even if the program is perfect.

4.2. Other errors detected at compilation time

Two main categories: file system problems and table overflow. Problems with the file system may be caused by protection (you may not read or create files) or by space problems (no space left on device; out of inodes; too many processes). Table overflow problems are often caused by peculiar source programs: very long procedures or functions, a lot of strings. Table overflow problems can sometimes be cured by giving a flag (-I or -sl) to pc [4].

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

4.3. 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 l-option is turned off, the compiler generates code to keep track of which source line causes which EM-1 instructions to be generated. It depends on the EM-1 implementation whether these LIN instructions are skipped or executed:

1. LIN instructions are always executed. The old line number is saved and restored whenever a procedure or function is called. All error messages contain this line number, except when the l-option was turned off.
2. same as above, but line numbers are not saved when procedures and functions are called.

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-VU 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-1 errors) can be found in [9]. Errors with trap number between 64 and 127 (Pascal errors) are listed in [9].

ISO 6.4.3.3: It shall be an error if any field-identifier defined within a variant is used in a field-designator unless the value of the tag-field is associated with that variant.

> This error is not detected. Sometimes this feature is used to achieve easy type conversion. However, using record variants this way is dangerous, error prone and not portable.

ISO 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 is done using the EM-1 array instructions. These instructions have three arguments: the array base address, the index and the address of the array descriptor. An array descriptor describes one dimension by three values: the element size, the lower bound on the index and the number of elements minus one. It depends on the EM-1 implementation whether these bounds are checked:
>
> 1. test on: checked (array bound error, trap 6, non-fatal).
>   test off: not checked
>
> 2. not checked.
>
> The other places where assignment-compatibility is required are:
>
> - assignment
> - value parameters
> - procedures read and readln
> - the final value of the for-statement
>
> For these places the compiler generates an EM-1 range check instruction, except when the r-option is turned off, or when the range of values of T2 is enclosed in the range of T1. If the expression consists of a single variable and if that variable is of a subrange type, then the subrange type itself is taken as T2, not its host-type. Therefore, a range instruction is only generated if T1 is a subrange-type and if the expression is a constant, an expression with two or more operands, or a single variable with a type not enclosed in T1. If a constant is assigned, then the EM-1 optimizer removes the range check instruction, except when the value is out of bounds.
>
> It depends on the EM-1 implementation whether the range check instruction is executed or skipped:
>
> 1. test on: checked (range bound error, trap 7, non-fatal).
>   test off: skipped

2. skipped

ISO 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.

ISO 6.5.4: It shall be an error if the pointer-variable has a nil-value or is undefined at the time it is de-referenced.

The EM-1 definition does not specify the binary representation of pointer values, so that it is not possible to choose an otherwise illegal binary representation for the pointer value NIL. Rather arbitrary the compiler uses the integer value zero to represent NIL. For all current implementations this does not cause problems.

The EM-1 definition does specify the size of pointer objects: 2 bytes. The compiler can be instructed, by the p-option, to use a different size for pointer objects. NIL is represented here by the appropriate number of zero words.

It depends on the EM-1 implementation whether de-referencing of a pointer with value NIL causes an error:

1. test on: for every de-reference the pointer value is checked to be legal. The value NIL is always illegal. Objects addressed by a NIL pointer always cause an error, except when they are part of some extraordinary sized structure (bad pointer, trap 26, fatal).
   test off: de-referencing for fetching will not cause an error to occur. However, if the pointer value is used for a store operation, a segmentation violation probably results (memory fault, trap 25, fatal). (Note: this is only true if the interpreter is executed with coinciding address spaces and protected text part. The interpreter must therefore be loaded with the '-n' option of the UNIX loader [5]).

2. de-referencing for a fetch operation will not cause an error. A store operation probably causes an error if the '-n' flag is specified to pc [4] or ld [5] while loading your program.

Some implementations of EM-1 initialize all memory cells for newly created variables with a constant that probably causes an error if that variable is not initialized with a value of its own type before use. For each implementation we give whether memory cells are initialized, with what value, and whether this value causes an error if de-referenced.

1. each memory word is initialized with the bit representation 1000000000000000, representing -32768 in 2's complement notation. For most small and medium sized programs this value will cause a segmentation violation (memory fault, trap 25,

fatal).

2. no initialization. Whenever a pointer is de-referenced, without being properly initialized, a segmentation violation (memory fault, trap 25, fatal) or 'bus error' are possible.

ISO 6.5.5: It shall be an error if the value of a file-variable f is altered while the buffer-variable is an actual variable parameter, or an element of the record-variable-list of a with-statement, or both.

This error is not detected

ISO 6.5.5: It shall be an error if the value of a file-variable f is altered by an assignment-statement which contains the buffer-variable f^ in its left-hand side.

This error is not detected.

ISO 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 f^ 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 full-proof.
- 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 IO 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.

ISO 6.6.5.3: It shall be an error to change any variant-part of a variable allocated by the form new(p,c1,...,cn) from the variant specified.

This error is not detected.

ISO 6.6.5.3: It shall be an error if a variable to be disposed had been allocated using the form new(p,c1,...,cn) with more variants specified than specified to dispose.

This error is not detected.

ISO 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.

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

The same comments apply as for de-referencing NIL or undefined pointers.

ISO 6.6.5.3: It shall be an error if a variable that is identified by the pointer parameter of dispose (or a component thereof) is currently either an actual variable parameter, or an element of the record-variable-list of a with-statement, or both.

This error is not detected.

ISO 6.6.5.3: It shall be an error if a referenced-variable created using the second form of new is used in its entirety as an operand in an expression, or as the variable in an assignment-statement or as an actual-parameter.

This error is not detected.

ISO 6.6.6.2: It shall be an error if the mathematical defined result of an arithmetic function would fall outside the set of values of the indicated result.

Except for the errors for undefined arguments, the following errors may occur for the arithmetic functions:

```
      abs(x):      none.
      sqr(x):      real underflow, trap 11, non-fatal;
                   real overflow, trap 10, non-fatal
      sin(x):      real underflow, trap 11, non-fatal
      cos(x):      real underflow, trap 11, non-fatal
      exp(x):      error in exp, trap 65, non-fatal (if x>10000);
                   real underflow, trap 11, non-fatal;
                   real overflow, trap 10, non-fatal
      ln(x):       error in ln, trap 66, non-fatal ( if x<=0)
      sqrt(x):     error in sqrt, trap 67, non-fatal (if x<0)
      arctan(x):   real underflow, trap 11, non-fatal;
                   real overflow, trap 10, non-fatal
```

ISO 6.6.6.2: It shall be an error if x in ln(x) is not greater than zero.

See above.

ISO 6.6.6.2: It shall be an error if x in sqrt(x) is negative.

See above.

ISO 6.6.6.2: It shall be an error if the integer value of trunc(x) does not exist.

This error is detected (real->int error, trap 17, non-fatal).

ISO 6.6.6.2: It shall be an error if the integer value of round(x) does not exist.

This error is detected (real->int error, trap 17, non-fatal).

ISO 6.6.6.2: 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.

ISO 6.6.6.2: It shall be an error if the character value of chr(x) does not exist.

Except when the r-option is turned off, the compiler generates an EM-1 range check instruction. The effect of this instruction depends on the EM-1 implementation as described before.

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

Same comments as for chr(x).

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

Same comments as for chr(x).

ISO 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).

ISO 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.

ISO 6.7.1: It shall be an error if any variable or function used as an operand in an expression is undefined at the time of its use.

Detection of undefined operands is only possible if there is at least one bit representation that is not allowed as legal value. The set of legal values depends on the type of the operand. To detect undefined operands, all newly created variables must be assigned a value illegal for the type of the created variable. The compiler itself does not generate code to initialize newly created variables. Instead, the compiler generates code to allocate some new memory cells. It is up to the EM-1 implementation to initialize these memory cells. However, the EM-1 machine does not know the types of the variables for which memory cells are allocated. Therefore, the best an EM-1 implementation can do is to initialize with a value that is illegal for the most common types of operands.

For all current EM-1 implementations we will describe whether memory cells are initialized, which value is used to initialize, for each operand type whether that value is illegal, and for all operations on all operand types whether that value is detected as undefined.

1. test on: new memory words are initialized with -32768. Assignment of this value is always allowed. Errors may occur whenever undefined operands are used in operations.

_______: -32768 is illegal. All arithmetic operations (except unary +) cause an error (undefined integer, trap 14, non-fatal). Relational operations do not, except for IN when the left operand is undefined. Printing of -32768 using write is allowed.
____: the bit representation of a real, caused by initializing the constituent memory words with -32768, is illegal. All arithmetic and relational operations (except unary +) cause an error (real undefined, trap 16, non-fatal). Printing causes the same error.
____: the value -32768 is illegal. For objects of type 'packed array[] of char' half the characters will have the value chr(0), which is legal, and the others will have the value chr(128), outside the valid ASCII range. The relational operators, however, do not cause an error.
_______: the value -32768 is illegal. For objects of type 'packed array[] of boolean' half the booleans will have the value false, while the others have the value v, where ord(v) = 128, naturally illegal. However, the Boolean and relational operations do not cause an error.
___: undefined operands of type set can not be distinguished from properly initialized ones. The set and relational operations, therefore, can never cause an error. However, if one forgets to initialize a set of character, then spurious characters like '/', '?', '0', '_' and 'o' appear.

test off: new memory cells are initialized with -32768. The only cases where this value causes an error are when an undefined operand of type real is used in an arithmetic or relational operation (except unary +) or when an undefined real is used as an argument to a standard function.

2. Newly created memory cells are not initialized and therefore they have a random value.

ISO 6.7.1: It shall be an error if the value of any member denoted by any member-designator of the set-constructor is outside the implementation-defined limits.

This error is detected (set bound error, trap 5, non-fatal).

ISO 6.7.1: It shall be an error if the possible types of an set-constructor do not permit it to assume a suitable type.

The compiler allocates as many bits as are necessary to store all elements of the host-type of the base-type of the set, not the base-type itself. Therefore, all possible errors can be detected at compile time.

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

It depends on the EM-1 implementation whether this error is detected:

1. test on: detected (divide by 0, trap 12, non-fatal).

test off: not detected.

2. not detected.

ISO 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 7i, non-fatal).

ISO 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.

The reaction depends on the EM-1 implementation:

1. test on: error detected if

   (result >= 32768) or (result < -32768).

   (integer overflow, trap 8, non-fatal). Note that if the result is -32768 the use of this value in further operations may cause an error.
   test off: not detected.

2. not detected.

ISO 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 4, fatal).

ISO 6.8.3.9: It shall be an error if the final-value of a for-statement is not assignment-compatible with the control-variable when the initial-value is assigned to the control-variable.

It is detected if the control variable leaves its allowed range of values while stepping from initial to final value. This is equivalent with the requirements if the for-statement is not terminated before the final value is reached.

ISO 6.9.2: 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).

ISO 6.9.2: 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).

ISO 6.9.2: It shall be an error if read is applied to f while f is undefined or not opened for reading.

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

ISO 6.9.4: It shall be an error if write is applied to f while f is un-defined or not opened for writing.

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

ISO 6.9.4: It shall be an error if TotalWidth or FracDigits as specified in write or writeln procedure calls are less than one.

This error is not detected. Moreover, it is considered an extension to allow zero or negative values.

ISO 6.9.6: It shall be an error if page is applied to f while f is unde-fined or not opened for writing.

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

5. Extensions to the standard

1. 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.

2. Assertions.

The Pascal-VU 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 off, 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.

3. 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.

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!

4. 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.

5. 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 -2147483647..+2147483647. Long constants may be declared. 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. 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. Two of these conversions may cause errors to occur:

```
real->longint error, trap 18, non-fatal
longint->int error, trap 19, non-fatal
```

This last error is only detected in implementation 1, with 'test on'. Note that all current implementations use target machine floating point instructions to perform some of the long operations.

6. Underscore as letter.

   The character '_' may be used in forming identifiers, if the u-option is turned on.

7. Zero field width in write.

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

8. Alternate symbol representation.

   The comment delimiters '(*' and '*)' are recognized and treated like '{' and '}'. The other alternate representations of symbols are not recognized.

6. Deviations from the standard

Pascal-VU deviates from the (March 1980) standard proposal in the following ways:

1. Only the first 8 characters of identifiers are significant, as requested by all standard proposals prior to March 1980. In the latest proposal, however, the sentence

   > "A conforming program should not have its meaning altered by the truncation of its identifiers to eight characters or the truncation of its labels to four digits."

   is missing.

2. The character sequences 'procedur', 'procedur8', 'functionXyZ' etc. are all erroneously classified as the word-symbols 'procedure' and 'function'.

3. Standard procedures and functions are not allowed as parameters in Pascal-VU, conforming to all previous standard proposals. 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;
   ```

4. The scope of identifiers and labels should start at the beginning of the block in which these identifiers or labels are declared. The Pascal-VU compiler, as most other one pass compilers, deviates in this respect, because the scope of variables and labels start at their defining-point.

7. 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 -.

f <num> the size of reals can be changed by this option. <num> should be specified in 16 bit words. The current default is 2, but might change to 4 in the future.

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, just fitting in one word on the PDP and many other minis.

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. These same line numbers are used for the profile, flow and count options of the EM-1 interpreter emi [6]. Default +.

p <num> the size of pointers can be changed by this option. <num> should be specified in 16 bit words. Default 1.

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. The compiler checks this flag just before the first symbol that follows the first 'begin' of the body of the procedure. Also, when the procedure exits, then the procedure 'procexit' is called if the t flag is on just before the last 'end' of the procedure body. Both 'procentry' and 'procexit' have a packed array of 8 characters as a parameter. Default procedures are present in the run time library. Default -.

u +/- if + then the character '_' is treated like a lower case letter, so that it may be used in identifiers. Procedure and function identifiers starting with an underscore may cause problems, because they may collide with library routine names. Default -.

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

A second method of passing options to the compiler ia available. This method uses the file on which the compact EM-1 code will be written. The compiler starts reading from this file scanning for options in the same format as used normally, except for the comment delimiters and the dollar sign. All options found on the file override the options set in your program. Note that the compact code file must always exist before the compiler is called.

The user interface program pc[4] takes care of creating this file normally and also writes one of its options onto this file. The user can specify, for instance, without changing any character in its Pascal program, that the compiler must include code for procedure/function tracing.

Another very powerful debugging tool is the knowledge that inaccessible statements and useless tests are removed by the EM-1 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.

A disadvantage of Pascal, the lack of preinitialized data, can be diminished by making use of the possibilities of the EM-1 optimizer. For instance, initializing an array of reserved words is sometimes optimized into 3 EM-1 instructions. To maximize this effect you must initialize variables as much as possible in order of declaration and array entries in order of increasing index.

8. References

[1] ISO standard proposal ISO/TC97/SC5-N462, dated February 1979. The same proposal, in slightly modified form, can be found in: A.M.Addyman e.a., "A draft description of Pascal", Software, practice and experience, May 1979. An improved version, received March 1980, is followed as much as possible for the current Pascal-VU.

[2] A.S.Tanenbaum, J.W.Stevenson, Hans van Staveren, "Description of an experimental machine architecture for use of block structured languages", Informatica rapport IR-54.

[3] W.S.Brown, S.I.Feldman, "Environment parameters and basic functions for floating-point computation", Bell Laboratories CSTR #72.

[4] UNIX manual pc(I).

[5] UNIX manual ld(I).

[6] UNIX manual emi(I).

[7] UNIX manual libpc(VII)

[8] UNIX manual pc_emlib(VII)

[9] UNIX manual pc_prlib(VII)