Pascal-VU reference manual

by

Johan W. Stevenson

(Dated May 29, 1979)

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

1. Introduction

This document refers to the ISO standard proposal for Pascal, ISO/TC97/SC5-N462 [1], dated february 1979. Pascal-VU complies with the requirements of this proposal as much as possible. 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 proposed standard and the list of options recognized by the compiler.

The Pascal-VU compiler produces code for an EM1 machine as defined in [2]. It is up to the implementor of the EM1 machine whether errors like integer overflow, undefined operand and range bound error are recognized or not. Therefore it depends on the EM1 machine implementation whether these errors are recognized in Pascal programs or not. For these errors the reaction of all known implementations is given.

There does not (yet) exist a hardware EM1 machine. Therefore, EM1 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 (using UNIX-6). Normally the interpreter performs some tests to detect undefined integers, integer overflow, range errors, etc. However, one of the options of the interpreter is to skip these tests. We will refer to these modes of operation as 'test on' and 'test off'.

2. a translator into PDP-11 instructions (using UNIX-6).

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 6.4.2.2 The predefined integer constant maxint, whose value shall be implementation-defined, shall define the subset of the integers available in any implementation over which the integer operations are defined.

The representation of integers in EM1 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 EM1 implementation:

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

2. -32768..+32767.

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

The format of reals is not defined in EM1. It is only defined that a real number occupies 4 bytes (32 bits) of storage. For each implementation of EM1 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

$$f = \pm f_1.b^{-1} + \ldots + f_p.b^{-p}$$

where

f_i = 0,...,epbase-1 and f_1 <> 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 EM1 implementations these constants are:

1. epbase = 2
   epprec = 24
   epemin = -127
   epemax = +127

2. idem ditto

ISO 6.4.2.2 The values of type char shall be an implementation-defined set of characters.

The 7-bits ASCII character set is used, where LF (10) denotes the end-of-line marker.

ISO 6.4.3.4 The largest and smallest values permitted in the base-type of a set-type shall be implementation-defined.

A value v is permitted if

0 <= ord(v) <= 255.

This restriction is posed by the Pascal compiler, not by the EM1 machine, which is able to support sets of 65536 bits easily. 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). So, the variable s, declared by

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

will contain 128 bits, not 10 or 16. 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 256. 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.

ISO 6.6.6.4 If the parameter of ord(x) is of type char, the result shall be implementation-defined.

The result of ord(x) for x of type char will be conform the ASCII character set:

0 <= ord(x) <= 127

ISO 6.7.2.2 The result of i DIV j for i < 0 and for j < 0 shall be implementation-defined.

The Pascal DIV operator is translated into an EM1 DIV instruction. Because EM1 is defined by an interpreter written in Pascal, there is a cycle. The current EM1-implementations, all for a PDP-11, use

the PDP DIV instruction to implement the EM1 DIV instruction. The Pascal Processor Handbook defines the DIV instruction such that the remainder is of the same sign as the dividend. Therefore:

```
 14 DIV  5 =  2        14 MOD  5 =  4
 14 DIV -5 = -3        14 MOD -5 = -1
-14 DIV  5 = -2       -14 MOD  5 = -4
-14 DIV -5 =  3       -14 MOD -5 =  1
```

ISO 6.9.4 The default field-width values for integer, Boolean and real types shall be implementation-defined.

The defaults are:

integer	6
Boolean	5
real	13

ISO 6.9.4 The number of digits written in an exponent part of a real shall be implementation-defined.

The number of digits in the exponent part is defined as

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

For the current implementations this evaluates to 2.

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 6.6.1 The full set of directives permitted after a procedure-heading shall be implementation-dependent.

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. Maintainance 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 EM1 assembly language. This is useful if you want to use some weird EM1 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 loaders task to check this. However, the current loaders are not that smart. Another solution is to check at runtime, at least the number of words for parameters. Some EM1-implementations check this:

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

2. not checked.

ISO 6.6.2 The full set of directives permitted after a function-heading shall be implementation-dependent.

The same as for procedures.

ISO 6.6.5.1 The effect of using standard procedures as procedural parameters shall be implementation-dependent.

It is not allowed to use standard procedures as procedural parameters.

ISO 6.6.5.2 If an activation of the procedure put(f) is not separated dynamically from a previous activation of get(f) or reset(f) by an activation of rewrite(f), the effect shall be implementation-dependent.

put(f) is only allowed when the file is opened for writing by a call to rewrite(f). get(f) is only allowed when the file is opened for reading for writing by a call to reset(f). Therefore appending to an existing file using the standard IO routines only is rather clumsy.

ISO 6.6.6.1 The effect of using standard functions as functional parameters shall be implementation-dependent.

It is not allowed to use standard functions as functional parameters. 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;
```

ISO 6.7.2.1 The order of evaluation of the operands of a binary operator shall be implementation-dependent.

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

1. the term

   ```
   factor1 / factor2
   ```

   where factor1 and factor2 are both of type integer is evaluated in the following steps:

   - evaluate factor1 into an integer result
   - evaluate factor2 into an integer result
   - convert factor1 to real
   - convert factor2 to real
   - divide

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 6.7.2.3 Whether a Boolean expression is completely or partially evaluated if its value can be determined by partial evaluation shall be implementation-dependent.

Boolean expressions are always evaluated completely, so the program part

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

is probably incorrect.

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 the selection of the variable in an assignment-statement involves the indexing of an array or the de-referencing of a pointer, 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 6.10 The effect of an explicit use of reset or rewrite on the standard files input and output shall be implementation-dependent.

rewrite(output) is a noop. reset(input) is equivalent to get(input). See also the chapter on deviations from the standard.

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 6.7.2.3 Whether a Boolean expression is completely or partially evaluated if its value can be determined by partial evaluation shall be implementation-dependent.

Boolean expressions are always evaluated completely, so the program part

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

is probably incorrect.

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 the selection of the variable in an assignment-statement involves the indexing of an array or the de-referencing of a pointer, 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 6.10 The effect of an explicit use of reset or rewrite on the standard files input and output shall be implementation-dependent.

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 consists of:

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

In all cases where the compiler detects an error which 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 file is read by the interface program pc [4], which 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 so constructed message 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 flavours, selected by flags given to pc [4]:

default: no listing, one line per error giving the line number and the error messages.

-e: for each erroneous line a listing of that line and its predecessor. The next line contains one or more characters '^' pinpointing 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.

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

4.3. Runtime errors

Errors detected at runtime 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 EM1 instructions to be generated. It depends on the EM1-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.

ISO 6.4.3.3 An error should be caused if a reference is made to a field of a variant other than the current 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 An error should be caused if an expression E of type T2 must be assignment-compatible with type T1, while T1 and T2 are compati-

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 consists of:

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

In all cases where the compiler detects an error which 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 file is read by the interface program pc [4], which 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 so constructed message 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 flavours, selected by flags given to pc [4]:

default: no listing, one line per error giving the line number and the error messages.

-e: for each erroneous line a listing of that line and its predecessor. The next line contains one or more characters '^' pinpointing 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.

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

4.3. Runtime errors

Errors detected at runtime 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 EM1 instructions to be generated. It depends on the EM1-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.

ISO 6.4.3.3 An error should be caused if a reference is made to a field of a variant other than the current 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 An error should be caused if an expression E of type T2 must be assignment-compatible with type T1, while T1 and T2 are compati-

ble ordinal-types and the value of E 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 EM1 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 EM1-implementation whether these bounds are checked:

1. **test on:** checked
   test off: not checked

2. not checked.

The other places where assignment-compatibility is required are:

- assignment
- value parameters
- procedures read and readln

For these places the compiler generates an EM1 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 the host-type of that subrange. 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 EM1 optimizer removes the range check instruction, except when the value is out of bounds.

It depends on the EM1-implementation whether the range check instruction is executed or skipped:

1. **test on:** checked
   test off: skipped

2. skipped

ISO 6.4.6 An error should be caused if an expression E of type T2 must be assignment-compatible with type T1, while T1 and T2 are compatible set-types and any of the members of the set expression E is not in the closed interval specified by the base-type of the type T1.

This error is not detected.

ISO 6.5.4 An error should be caused if the pointer value is NIL or undefined at the time it is de-referenced.

The EM1 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 EM1 definition does specify the size of pointer objects: 2 bytes. However, when the v-option is turned on, the compiler produces code for one of the planned successors of EM1, a machine with 32 bits pointer values. In this case NIL is represented by 32 zero bits.

It depends on the EM1-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 means of a NIL pointer always cause an error, except when they are part of some extraordinary sized structure. **test off:** de-referencing for the purpose of fetching will not cause an error to occur. However, if the pointer value is used for a store operation, a segmentation violation probably results. (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 EM1 initialize all memory cells for newly created variables with a constant likely to cause 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 which 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.

2. no initialization. Whenever a pointer is de-referenced, without being properly initialized, a 'segmentation violation' or a 'bus error' are possible.

ISO 6.6.5.2 An error should be caused if eof(f) does not yield true prior to execution of put(f).

This error is detected indirectly. Execution of put(f) will cause an error if f is not opened for writing by using rewrite(f). Rewrite(f) causes eof(f) to yield true. All standard procedures allowed for files opened for writing, including put(f), do not

cause eof(f) to yield false.

ISO 6.6.5.2 An error should be caused if eof(f) does not yield false prior to execution of get(f).

This error is detected.

ISO 6.6.5.2 An error should be caused if the current file position of a file is altered while the buffer variable f^ 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 6.6.5.3 An error should be caused if the value of the pointer parameter of dispose is NIL or undefined.

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

ISO 6.6.5.3 An error should be caused if a variable that is currently either an actual variable parameter, or an element of the record-variable-list of a with-statement, or both, is referred to by the pointer parameter of dispose.

This error is not detected.

ISO 6.6.5.3 An error should be caused if a referenced-variable created using the second form of new is used as an operand in an expression, or the variable in an assignment-statement or as an actual-parameter.

This error is not detected.

ISO 6.6.6.2 An error should be caused if x not greater than zero in ln(x).

This error is detected.

ISO 6.6.6.2 An error should be caused if x is negative in sqrt(x).

This error is detected.

ISO 6.6.6.2 An error should be caused if trunc(x) is not a value of type integer.

This error is detected.

ISO 6.6.6.2 An error should be caused if round(x) is not a value of type integer.

This error is detected.

ISO 6.6.6.2 An error should be caused if chr(x) does not exist.

Except when the r-option is turned off, the compiler generates an

EM1 range check instruction. The effect of this instruction depends on the EM1-implementation as described before.

ISO 6.6.6.2 An error should be caused if succ(x) does not exist.

Same comments as for chr(x).

ISO 6.6.6.2 An error should be caused if pred(x) does not exist.

Same comments as for chr(x).

ISO 6.7.1 An error should be caused if any variable or function used as an operand in an expression has an undefined value at the time of its use.

Detection of undefined operands is only possible if there is at least one bit representation which 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 EM1-implementation to initialize these memory cells. However, the EM1 machine does not know the types of the variables for which memory cells are allocated. Therefore, the best an EM1-implementation can do is to initialize with a value which is likely to be illegal for the most common types of operands.

For all current EM1-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 that value is always allowed. Errors may occur whenever undefined operands are used in operations.
   integer: -32768 is illegal. All arithmetic operations (except unary +) cause an error. Relational operations do not, except for IN when the left operand is undefined. Printing of -32768 using write is allowed.
   real: 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. Printing causes an error.
   char: 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.
   Boolean: 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.
set: 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 one of the standard functions.

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

ISO 6.7.1 An error should be caused if the value of an expression which is the member of a set is outside the implementation-defined limits.

This error is detected.

ISO 6.7.2.2 An error should be caused if j is zero in i DIV j.

It depends on the EM1-implementation whether this error is detected:

1. **test on:** detected.
   test off: not detected.

2. not detected.

ISO 6.7.2.2 An error should be caused if the result of a binary operation on integer operands is not in the range -maxint..+maxint.

The reaction depends on the EM1-implementation:

1. **test on:** error detected if

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

   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 An error should be caused if none of the case-constants is equal to the current value of the selector.

This error is detected.

ISO 6.8.3.9 An error should be caused if the control-variable is assigned to by the repeated statement or altered by any procedure or func-

tion activated by the repeated statement.

This error is not detected.

ISO 6.9.2 An error should be caused 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.

ISO 6.9.2 An error should be caused 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.

5. Extensions to the standard

1. The compiler is able to compile modules, not forming complete programs, but consisting of procedures and functions which can be used to form libraries. The syntax of these modules is

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

The compiler accepts a program or a module:

```
unit = program | module
```

It is only allowed to declare variables local to the procedures and functions. All variables outside these procedures and functions must be imported and exported by parameters, even the files input and output. By giving the correct procedure/function heading followed by the directive 'extern' you may use procedures and functions declared in other units.

2. 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. 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.

release:
mark: for most applications it is sufficient to use the heap as second stack. Mark and release are suited for this kind of use, more suited than dispose. mark(p), with p of

type pointer, stores the current value of the heappointer in p. release(p), with p initialized by a call of mark(p), restores the heappointer 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. If the c-option is turned on, then some special features are available to facilitate 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 dereferenced. Still these strings are very useful in combination with external routines.

5. The character '_' may be used in forming identifier, except as first character.

6. The comment delimiters '{' and '}' may be nested.

7. Comments delimited by '(*' and '*)' are recognized as well, but may not be nested.

6. Deviations from the standard

There is one serious defect in the standard proposal (and in the language as described by Jensen and Wirth). This defect causes all implementors of Pascal on machines with a time-sharing environment to deviate from the standard. The defect is that interactive input is impossible. More precisely: it is not possible to prompt a user that the program is running and requires input, because the file window input^ must be initialized with the first character of input before any other statement can be executed. This is caused by the requirement of the standard that the file input, if mentioned in the program heading, must be initialized by the statement reset(input).

The solution we adopted is to replace this requirement by:

> The file input, if mentioned in the program heading, is opened such that:
>
> - the statement reset(input) is not required to start reading,
> - the file window input^ is initialized with a blank,
> - the function eoln(input) yields true.
>
> Moreover, the statement reset(input) is equivalent to get(input) as stated before.

This solution offers several advantages:

- Interactive input is possible with a minimal change to the language.
- Most standard programs will work without any modification. Only if programs skip leading spaces, problems may occur.
- If programs conforming to the standard must be modified, then this modification is trivial: add the statement get(input).
- Programs created on the Pascal-VU system can be ported easily to other installations. They will work without any modification, if an empty line is prepended to the input. Modification is probably trivial: removal of the statement get(input).
- The first character of the input file can be obtained in three different ways:
  get(input),
  readln(input) and
  reset(input).

We present two examples demonstrating why the alternatives for initializing the window input^ are useful. The first of these alternatives, get(input), is probably best for portability reasons. However, a call of readln(input) makes reading of all lines similar, as is demonstrated by the following program:

```
program add(input,output);
{This program adds integers}
var stop:boolean;
    i,total:integer;
begin total:=0; stop:=false;
    repeat
```

```
            write('next number: ');
            readln;
            if eof then
                stop:=true
            else
                begin
                    read(i);
                    total:=total+i
                end
        until stop;
        writeln('these numbers add to ',total:1)
    end.
```

Transporting this program to a fully standard Pascal system is more difficult. Initializing the file window by reset(input) may be used in programs like:

```
    program append(input,secondin,output);
    {This program concatenates two files}
    var secondin:text;

    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}
        copy(input);
        copy(secondin)
    end.
```

7. Compiler options

Some options of the compiler may be controlled by using "{$....}" or "(*$....*)". 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 runtime. 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 -.

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 a lot of 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 EM1 interpreter em1 [6]. 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. 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 library libP [7]. Default -.

v +/- if + then 32 bit addresses are produced for an EM1 machine with 256 data segments of 64k bytes each. Default -.

Three of these flags (c,i and v) are only effective when they appear before the 'program' symbol. The others may be switched on and off.

Instead of including the options in your Pascal program, you may also set them by giving a flag argument with the same syntax to pc [4]. Pc passes this flag to the compiler by using the intermediate file which is also used, in reverse direction, for the error messages in condensed form. Options given to pc override the options in your program. This feature is very useful for debugging. Without changing any character in

your program you may, for instance, include code for procedure/function tracing.

Another very powerful debugging tool is the knowledge that inaccessable statements and useless tests are removed by the EM1 optimizer. For instance, a statement like:

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

is completely removed by the optimizer when 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.

One of the disadvantages of Pascal, the lack of preinitialized data, can be diminished by making use of the possibilities of the EM1 optimizer. For instance, initializing an array of reserved words is sometimes optimized into 3 EM1 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.

[2] A.S.Tanenbaum, J.W.Stevenson, J.M.van Staveren, "Description of an experimental machine architecture for use of block structured languages", Informatika rapport IR-??. (To appear).

[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 em1(I).

[7] UNIX manual libP(VII).