Reading Textfiles with New Stanford Pascal

In the following paper, I will try to describe how reading textfiles in Stanford Pascal should work. That is, the paper does not describe the current situation, but instead it describes the situation as it should be in my opinion. Later, I will do corrective actions to the different runtimes to adjust them, that is, to bring the runtimes nearer to the situation as it should be in my opinion.

The motiviation for all this is: I observed that a test program I wrote to test the new CSP RDV (standard procedure to read variable length strings) did strange things.

More precisely: when I read varying strings in a loop and the READ calls in the loop (CSP RDV) encountered an EOLN condition, the program entered an endless loop, if I didn‘t a READLN call to read a new line. The same is true if I read fixed length strings (CSP RDS).

But this problem does not occur, if I use single character reads (CSP RDC). In this case, at EOLN a single blank is returned and reading continues without problems with the next line.

That is, the following program part reads the whole input file successfully until EOF:

```
var INPFILE : TEXT ;
    CH : CHAR ;
    CH10 : CHAR ( 10 ) ;
    S10 : STRING ( 10 ) ;

begin (* HAUPTPROGRAMM *)
  WRITELN ;
  WRITELN ( 'lesen einzelne char' ) ;
  WRITELN ;
  RESET ( INPFILE ) ;
  while not EOF ( INPFILE ) do
    begin
      READ ( INPFILE , CH ) ;
      WRITELN ( 'gelesen: ' , CH , ORD ( CH ) : 4 ,
                ' eof = ' , EOF ( INPFILE ) ,
                ' eoln = ' , EOLN ( INPFILE ) ) ;
    end (* while *) ;
```

In contrast, a similar program part which uses a CHAR (10) or STRING (10) variable to read the input textfile will only work, if you do an explicit READLN whenever you encounter an EOLN condition:

```
var INPFILE : TEXT ;
    CH : CHAR ;
    CH10 : CHAR ( 10 ) ;
    S10 : STRING ( 10 ) ;

begin (* HAUPTPROGRAMM *)
  WRITELN ;
  WRITELN ( 'lesen char (10)' ) ;
  WRITELN ;
  RESET ( INPFILE ) ;
  while not EOF ( INPFILE ) do
    begin
      READ ( INPFILE , CH10 ) ;
      WRITELN ( 'gelesen: <' , CH10 ,
                '> eoln = ' , EOLN ( INPFILE ) ) ;
      if EOLN ( INPFILE ) then
        READLN ( INPFILE ) ;
    end (* while *) ;
```

this seems to be plain wrong; the loop should work without this IF EOLN THEN READLN coding and not be an endless loop.

I will try to describe in the following paper in detail, how the CSPs RDC, RDS and RDV should work on the different platforms in different circumstances, this way this document should serve as a guideline and reference for the implementation. Remember: the CSPs have to be implemented two times, first in the interpreter PCINT.C for the non-mainframe platforms (C functions called cspf_xxx, xxx being the name of the CSP), and second on the mainframe in the Pascal monitor PASMONN ASSEMBLE.

Furthermore, I am thinking about implementing the other CSPs which implement READ functions (e.g. RDI for integer, RDR for real etc.) in Pascal, so that there is a single portable implementation which is guaranteed to work exactly the same on both platforms. Another idea is to add length information to READ statements, like in IBM‘s Pascal/VS, which involves major extensions to the READ CSPs. And: implementing READSTR and WRITESTR.

The paper is organized as follows:

First a paragraph about **Reading single characters (implemented by the CSP RDC).**

Then a discussion about **Terminal textfiles** (what they are and how they are different to normal textfiles).

The next paragraph describes **Reading strings of varying length (using the CSP RDV).**

Very similar is **Reading strings of fixed length (using the CSP RDS).**

Finally there is a paragraph discussing **Reading in binary mode (without translation)**; translation in this case does NOT refer to code conversion, but only to the translation of the different line end representations (different between platforms) to the Pascal logic (EOLN function; blanks are returned at line end).

Some additional explanations and remarks:

- The reader should be familiar with some Pascal basics and he or she should know the standard procedures of the Pascal runtime, like READ, READLN, GET etc.

- CSP (= call standard procedure) is a P-Code instruction, which the Stanford Pascal compiler generates for every call to a standard procedure; CSP RLN for example = READLN.

- P-Code, BTW, is an intermediate ASSEMBLER-like code for a hypothetical stack machine which is able to execute P-Code directly. P-Code stands for portable code (not Pascal-Code). It is interpreted on Non-mainframe platforms by an ANSI C program called PCINT (P-Code interpreter). PCINT is a sort of software implementation of the P-machine. On the mainframe, the P-Code Source (which is a textfile, after all) is translated by PASCAL2 (the second pass of Stanford Pascal) to a 370 machine code executable.

Reading single characters (CSP RDC)

Reading of single characters is done via the CSP RDC. The CSP RDC is generated by the compiler when a single char variable is read from a textfile.

There are several issues with textfiles:

- textfiles are implemented on the mainframe using the mainframe file system; that is, the record structure is enforced or implemented by the operating system. The files have variable or fixed record format (RECFM F or V) and the fixed or maximum record length is part of the file description

- in contrast, textfiles on the other platforms are simple byte strings (as any other – binary – files, BTW)

- lines of textfiles on the mainframe are identical to records of files (see above); on the other platforms, there is no concept of „records“ with text files. Instead the lines are implemented by line end characters or character sequences (like 0x0a on Unix or 0x0d0a on Windows)

- these differences in textfile representation are hidden by the Pascal runtime

There is one minor difference which cannot be resolved:

- on the non-mainframe platforms it is possible to have lines of length zero (two line end sequences immediately following each other)

- in contrast, on the mainframe, each line has a minimum length of 1; this is true even if you do two WRITELNs with nothing in between on an output file with RECFM V. The resulting „empty“ line has one space character on the mainframe. (I am not sure if this a requirement of the mainframe operating systems, but this was true even for VB files which were not created by New Stanford Pascal – e.g. files created by the System Editor on VM/CMS).

Now, when reading textfiles using the CSP RDC (read character), it will work as follows (on all platforms):

- READ (F, C) will in general read the next character from the file F into the character variable C
- if there are remaining characters in the same line after the character which has just been read, EOLN (F) will be false, otherwise (if the character just read was the last char in the line), EOLN (F) will be true.
- If another READ (F, C) is issued after EOLN (F), a space character will be returned, EOLN (F) will be false, and the next line will be prepared for processing. This is only true, if there is another line; otherwise EOF (F) will be true and the result of READ (F, C) should be ignored.

See the following sample program:

```
program TESTRDC ( INPFILE , OUTPUT ) ;

//****************************************
// Testen RDC
//****************************************

var INPFILE : TEXT ;
    CH : CHAR ;

begin (* HAUPTPROGRAMM *)
  RESET ( INPFILE ) ;
  READ ( INPFILE , CH ) ;
  while not EOF ( INPFILE ) do
    begin
      WRITELN ( 'gelesen: ' , CH , ORD ( CH ) : 4 ,
                ' eof = ' , EOF ( INPFILE ) ,
                ' eoln = ' , EOLN ( INPFILE ) ) ;
      READ ( INPFILE , CH ) ;
    end (* while *) ;
end (* HAUPTPROGRAMM *) .
```

It is important that EOF (F) is set to true when CSP RDC tries to read another line after the last line. Any call of CSP RDC after that will throw a BADIO exception.

Instead of simply READing across line ends like in the example above, you can issue READLN whenever you found an EOLN condition. See the example below.

But: it is important that in this case READLN prepares the next line, and so the test for EOF must be made after the READLN call. If the EOF condition is true after READLN, no more READ call is possible (otherwise: BADIO exception).

```
WRITELN ;
WRITELN ( 'lesen mit read und readln' ) ;
WRITELN ;
RESET ( INPFILE ) ;
while TRUE do
  begin
    if EOF ( INPFILE ) then
      break ;
    if EOLN ( INPFILE ) then
      begin
        READLN ( INPFILE ) ;
        WRITELN ( '-------------------------------------------' ) ;
        continue ;
      end (* then *) ;
    READ ( INPFILE , CH ) ;
    WRITELN ( 'gelesen: ' , CH , ORD ( CH ) : 4 ,
              ' eof = ' , EOF ( INPFILE ) ,
              ' eoln = ' , EOLN ( INPFILE ) ) ;
  end (* while *) ;
```

You should also note that this example takes special care about lines with zero length; that‘s why there is the CONTINUE statement after the READLN - WRITELN sequence. If you would do a READ after a READLN, and if the READLN had prepared a zero length line for input, the READ (of a single char) would return a space character and prepare the next line for input.

It is of course possible to call READLN before reaching the end of a line.

For example: consider the following program part, where only the first 10 characters of every line are needed. It should be possible that a line contains less than 10 characters; the program should deal with this situation.

```
WRITELN ;
WRITELN ( 'lesen mit read und readln - max 10 cols' ) ;
WRITELN ;
RESET ( INPFILE ) ;
COLS := 0 ;
while TRUE do
  begin
    if EOF ( INPFILE ) then
      break ;
    if EOLN ( INPFILE ) or ( COLS >= 10 ) then
      begin
        READLN ( INPFILE ) ;
        WRITELN ( '-------------------------------------------' ) ;
        COLS := 0 ;
        continue
      end (* then *) ;
    READ ( INPFILE , CH ) ;
    WRITELN ( 'gelesen: ' , CH , ORD ( CH ) : 4 ,
              ' file char: ' , INPFILE -> , ORD ( INPFILE -> ) : 4 ,
              ' eof = ' , EOF ( INPFILE ) ,
              ' eoln = ' , EOLN ( INPFILE ) ) ;
    COLS := COLS + 1 ;
  end (* while *) ;
```

The program keeps track of the characters already read in the current line (in the variable COLS), and if this variable reaches the value 10, a READLN - WRITELN sequence is scheduled.

A general note:

when working with textfiles, the file variable (INPFILE → in the examples above) always contains the next character which will be read by the next READ call. You can use this to decide if the next character in the buffer will be interesting. When EOLN is true, the next character will always be blank.

See example above (INPFILE → is printed).

Terminal textfiles

In Stanford Pascal reading stuff from textfiles is also used to do terminal (a.k.a. console) I/O.

Pascal I/O, as defined by the Pascal Standard, and console I/O do not go together well. For example:

- for every file opened for input, an implicit RESET call is done at file open time
- this implicit RESET call reads the first buffer from the file (the file variable F → is defined after the RESET and contains the first element of the file)
- this means, that the first line of an input console file would have to be entered by the user at the very beginning of the program, even before a WRITELN to write an input prompt could have been issued.

To solve this problem, I inserted a terminal flag in the Pascal FCB (file control block), which is set to N normally, but is set to Y in case of a terminal (console) file. And I deferred all implicit RESETs and REWRITEs until first READ or WRITE.

You should not issue RESET on terminal files, because this would force the user to enter a line. In a similar way I don‘t recommend to use GET; GET on terminal files should be restricted to the Pascal runtime. Simply use READ and READLN to read information from terminal files. The first READ will implicitly do the RESET (and this way read the first buffer).

There are some more differences between „normal“ file I/O and terminal I/O:

- terminal I/O never implicitly issues READLN to retrieve a new input buffer (because this would involve a new prompt for input from the terminal user)
- instead in this case READ after EOLN would always return spaces
- even if you do READLN, the prompt for a new input buffer is deferred until the next read

This different behaviour allows the application to better control the terminal I/O.

The terminal flag in the Pascal FCB is set according to the following rules:

- it is set to N normally
- but: it is set to Y for the files INPUT and OUTPUT, if there are no allocations for INPUT and OUTPUT involving disk files (using environment variables or similar mechanisms)
- and: it is set to Y for other files, if there is a TERMIN or TERMOUT call (see PASUTILS PASCAL extension).

See the example program on the following page. The program shows how to read input lines from a terminal textfile (file INPFILE set to terminal using TERMIN).

There is a problem using single character READs in this case; you will not be able to distinguish an empty input from the input of a single blank, because in both cases EOLN will be shown after the first char returned (which is blank in both cases).

Therefore it is much better to use the CSP RDV (read into varying string), which is shown in the example, too. When doing this, you can retrieve the length of the input from the string length. The READLN used in this example will ignore longer input (in this case, the string variable is defined as STRING (10)).

Sample program using terminal I/O:

```
program TESTRDC2 ( INPFILE , OUTPUT ) ;

//****************************************
// Testen RDC
//****************************************


var INPFILE : TEXT ;
    CH : CHAR ;
    COLS : INTEGER ;
    DONE : BOOLEAN ;
    S10 : STRING ( 10 ) ;

begin (* HAUPTPROGRAMM *)
  WRITELN ( 'input lines until line contains $' ) ;
  TERMIN ( INPFILE ) ;
  DONE := FALSE ;
  repeat
    for COLS := 1 to 10 do
      begin
        READ ( INPFILE , CH ) ;
        WRITELN ( 'found: ' , CH , ORD ( CH ) : 4 ,
                  ' eof = ' , EOF ( INPFILE ) ,
                  ' eoln = ' , EOLN ( INPFILE ) ) ;
        if CH = '$' then
          DONE := TRUE
      end (* for *) ;
    READLN ( INPFILE ) ;
  until DONE ;
  WRITELN ( 'input strings until string contains $' ) ;
  DONE := FALSE ;
  repeat
    READLN ( INPFILE , S10 ) ;
    WRITELN ( 'found: ' , LENGTH ( S10 ) : 4 , ' <' , S10 , '>' ) ;
    DONE := INDEX ( S10 , '$' ) <> 0 ;
  until DONE ;
end (* HAUPTPROGRAMM *) .
```

Some sample output from program TESTRDC2:

from the first loop:

```
input lines until line contains $
oppolzer
found: o 111 eof = FALSE eoln = FALSE
found: p 112 eof = FALSE eoln = FALSE
found: p 112 eof = FALSE eoln = FALSE
found: o 111 eof = FALSE eoln = FALSE
found: l 108 eof = FALSE eoln = FALSE
found: z 122 eof = FALSE eoln = FALSE
found: e 101 eof = FALSE eoln = FALSE
found: r 114 eof = FALSE eoln =  TRUE
found:    32 eof = FALSE eoln =  TRUE
found:    32 eof = FALSE eoln =  TRUE
```

from the second loop:

```
input strings until string contains $
bernd
found:    5 <bernd>
oppolzer
found:    8 <oppolzer>

found:    0 <>

found:    1 < >
12345678901234567890
found:   10 <1234567890>
xx$
found:    3 <xx$>
```

Reading strings of varying length (CSP RDV)

The example program in the last paragraph showed how the CSP RDV (READ with strings of varying length) is used to process textfiles. This is – of course – not only possible for terminal files, but for normal disk files, too.

See for example the following program, which copies a file to another file.

```
program TESTCPY2 ( OUTPUT , INP1 , OUT2 ) ;

var INP1 : TEXT ;
    OUT2 : TEXT ;
    CV : STRING ( 80 ) ;
    ZEILNR : INTEGER ;
    LAENGE : INTEGER ;
    LMAX : INTEGER ;
    COUT : STRING ( 80 ) ;

begin (* HAUPTPROGRAMM *)
  RESET ( INP1 ) ;
  REWRITE ( OUT2 ) ;
  ZEILNR := 0 ;
  while not EOF ( INP1 ) do
    begin
      READLN ( INP1 , CV ) ;
      ZEILNR := ZEILNR + 1 ;
      LMAX := MAXLENGTH ( CV ) ;
      LAENGE := LENGTH ( CV ) ;
      WRITELN ( 'Zeile ' , ZEILNR , ' MaxL ' , LMAX , ' Laenge ' ,
                LAENGE ) ;
      COUT := RTRIM ( CV ) ;
      WRITELN ( 'Laenge der Ausgabezeile = ' , LENGTH ( COUT ) ) ;
      WRITELN ( OUT2 , COUT )
    end (* while *) ;
  CLOSE ( OUT2 ) ;
end (* HAUPTPROGRAMM *) .
```

On the PC, when this program is used to read its own source code, the empty lines are reported to have line length zero … and they are reproduced in the same way in the output file. The output file has exactly the same byte size (reported by Windows) as the source file. (Important: this is only true, because the original file has no trailing blanks … otherwise the RTRIM call would remove the trailing blanks and the file size would be different).

Another possibility is: don‘t use READLN, but use a sequence of simple READ calls for varying strings instead.

Let‘s, for example, assume a STRING (10) variable S10 and a procedure call READ (F, S10).

This will work as follows:

- The input lines will be returned in pieces of 10 chars, until a line end is found; if a line end is found, the resulting string may be shorter than 10 and EOLN will be set to true. It will not be zero, unless the whole input line has zero length, that is: if a line has 50 characters in this case, the first 4 calls will return 10 chars and EOLN = false, the 5th call will return 10 chars and EOLN = true.

- In contrast, if the line had 47 chars, the 5th call would return 7 chars and EOLN = true.

- The next call after EOLN would prepare the next line for processing and return the first 10 characters.

- EOLN would be false, if there are more than 10 characters in the second line

- otherwise EOLN would be true

- if the second line had length zero (which is not possible on the mainframe), EOLN would be true and S10 would have zero length.

Example program:

```
program TESTRDV ( INPFILE , OUTPUT ) ;

var INPFILE : TEXT ;
    S10 : STRING ( 10 ) ;

begin (* HAUPTPROGRAMM *)
  RESET ( INPFILE ) ;
  READ ( INPFILE , S10 ) ;
  while not EOF ( INPFILE ) do
    begin
      WRITELN ( 'gelesen: <' , S10 , '> eoln = ' , EOLN ( INPFILE ) ) ;
      READ ( INPFILE , S10 ) ;
    end (* while *)
end (* HAUPTPROGRAMM *) .
```

If the program above is used to process its own source code, the output is as follows:

```
gelesen: <program TE> eoln = FALSE
gelesen: <STRDV ( IN> eoln = FALSE
gelesen: <PFILE , OU> eoln = FALSE
gelesen: <TPUT ) ;> eoln =  TRUE
gelesen: <> eoln =  TRUE
gelesen: <> eoln =  TRUE
gelesen: <var INPFIL> eoln = FALSE
gelesen: <E : TEXT ;> eoln =  TRUE
gelesen: <    S10 : > eoln = FALSE
gelesen: <STRING ( 1> eoln = FALSE
gelesen: <0 ) ;> eoln =  TRUE
gelesen: <> eoln =  TRUE
gelesen: <> eoln =  TRUE
gelesen: <begin (* H> eoln = FALSE
gelesen: <AUPTPROGRA> eoln = FALSE
gelesen: <MM *)> eoln =  TRUE
gelesen: <  RESET ( > eoln = FALSE
gelesen: <INPFILE ) > eoln = FALSE
gelesen: <;> eoln =  TRUE
gelesen: <  READ ( I> eoln = FALSE
gelesen: <NPFILE , S> eoln = FALSE
gelesen: <10 ) ;> eoln =  TRUE
gelesen: <  while no> eoln = FALSE
gelesen: <t EOF ( IN> eoln = FALSE
gelesen: <PFILE ) do> eoln =  TRUE
gelesen: <    begin> eoln =  TRUE
gelesen: <      WRIT> eoln = FALSE
gelesen: <ELN ( 'gel> eoln = FALSE
gelesen: <esen: <' ,> eoln = FALSE
gelesen: < S10 , '> > eoln = FALSE
gelesen: <eoln = ' ,> eoln = FALSE
gelesen: < EOLN ( IN> eoln = FALSE
gelesen: <PFILE ) ) > eoln = FALSE
gelesen: <;> eoln =  TRUE
gelesen: <      READ> eoln = FALSE
gelesen: < ( INPFILE> eoln = FALSE
gelesen: < , S10 ) ;> eoln =  TRUE
gelesen: <    end (*> eoln = FALSE
gelesen: < while *)> eoln =  TRUE
gelesen: <end (* HAU> eoln = FALSE
gelesen: <PTPROGRAMM> eoln = FALSE
gelesen: < *) .> eoln =  TRUE
```

Reading strings of fixed length (CSP RDS)

Remember: fixed length strings are variables which have the type ARRAY [1 .. N] OF CHAR, but this type may be abbreviated as CHAR (N) in Stanford Pasal (N is a constant).

Reading fixed length strings works much the same way as reading varying strings. But of course there is the drawback that with fixed length strings you are not able to retrieve the precise length of the input line, because fixed length strings will be padded with blanks at the right (if the input line is shorter).

BTW: this is what the compiler uses for the input of the source lines. Variable strings were not yet implemented when the compiler was first created.

For the discussion of the CSP RDS, it is IMO sufficient to do a minor change to the sample programs from the last paragraph.

```
program TESTCPY3 ( OUTPUT , INP1 , OUT1 ) ;

var INP1 : TEXT ;
    OUT1 : TEXT ;
    CF : CHAR ( 80 ) ;
    ZEILNR : INTEGER ;
    LAENGE : INTEGER ;
    LMAX : INTEGER ;

begin (* HAUPTPROGRAMM *)
  RESET ( INP1 ) ;
  REWRITE ( OUT1 ) ;
  ZEILNR := 0 ;
  while not EOF ( INP1 ) do
    begin
      READLN ( INP1 , CF ) ;
      ZEILNR := ZEILNR + 1 ;
      LMAX := MAXLENGTH ( CF ) ;
      LAENGE := LENGTH ( CF ) ;
      WRITELN ( 'Zeile ' , ZEILNR , ' MaxL ' , LMAX ,
                ' Laenge ' , LAENGE ) ;
      WRITELN ( OUT1 , CF )
    end (* while *) ;
  CLOSE ( OUT1 ) ;
end (* HAUPTPROGRAMM *) .
```

The program shown above will write a copy of the input file INP1 to output file OUT1, but all lines will be padded with blanks to a length of 80 characters. (BTW: if there are lines longer than 80 characters, all characters after position 80 are ignored, due to the READLN call on INP1).

Another example is a program similar to TESTRDV from the previous paragraph, which does no READLN, but reads the whole input file using only READ calls involving a variable C20 which is a fixed char string of length 20.

```
program TESTRDS ( INPFILE , OUTPUT ) ;

var INPFILE : TEXT ;
    C20 : CHAR ( 20 ) ;

begin (* HAUPTPROGRAMM *)
  RESET ( INPFILE ) ;
  READ ( INPFILE , C20 ) ;
  while not EOF ( INPFILE ) do
    begin
      WRITELN ( 'gelesen: <' , C20 , '> eoln = ' , EOLN ( INPFILE ) ) ;
      READ ( INPFILE , C20 ) ;
    end (* while *)
end (* HAUPTPROGRAMM *) .
```

and the output of this program, given its own source code as input, is:

```
gelesen: <program TESTRDS ( IN> eoln = FALSE
gelesen: <PFILE , OUTPUT ) ;  > eoln =  TRUE
gelesen: <                    > eoln =  TRUE
gelesen: <var INPFILE : TEXT ;> eoln =  TRUE
gelesen: <    C20 : CHAR ( 20 > eoln = FALSE
gelesen: <) ;                 > eoln =  TRUE
gelesen: <                    > eoln =  TRUE
gelesen: <begin (* HAUPTPROGRA> eoln = FALSE
gelesen: <MM *)               > eoln =  TRUE
gelesen: <  RESET ( INPFILE ) > eoln = FALSE
gelesen: <;                   > eoln =  TRUE
gelesen: <  READ ( INPFILE , C> eoln = FALSE
gelesen: <20 ) ;              > eoln =  TRUE
gelesen: <  while not EOF ( IN> eoln = FALSE
gelesen: <PFILE ) do          > eoln =  TRUE
gelesen: <    begin           > eoln =  TRUE
gelesen: <      WRITELN ( 'gel> eoln = FALSE
gelesen: <esen: <' , C20 , '> > eoln = FALSE
gelesen: <eoln = ' , EOLN ( IN> eoln = FALSE
gelesen: <PFILE ) ) ;         > eoln =  TRUE
gelesen: <      READ ( INPFILE> eoln = FALSE
gelesen: < , C20 ) ;          > eoln =  TRUE
gelesen: <    end (* while *) > eoln =  TRUE
gelesen: <end (* HAUPTPROGRAMM> eoln = FALSE
gelesen: < *) .               > eoln =  TRUE
```

Reading in binary mode (without translation)

For special applications, it might be interesting to read files in binary mode, without any translation done by the Pascal runtime. This is possible, if the file used in the Pascal program is NOT defined as a FILE OF CHAR (or TEXT), but as a file of another type instead (for example a record structure which may contain a character). A working example follows.

Some remarks:

- Pascal never does code conversion (ASCII to EBCDIC or vice versa) implicitely; the character set is always determined by the target machine
- the ORD function is used to get the numeric value of the characters in the current character set; its value is always non-negative
- if you use the mechanism mentioned above to do binary reads, your programs will normally be non-portable (because, for example, CR LF sequences are read on Windows, but not on mainframe platforms, because they don‘t exist there)
- that is: „without translation“ in the paragraph above only refers to the translation of line ends to spaces etc. - because there are no other translations, esspecially no implicit codepage translations (on all platforms)

The sample program on the next page uses READ (F, X) to read the structures from the (binary) file, although it is more common to use GET (F) and refer to the file variable F →, when doing binary I/O.

READ (F, X) with binary files is implemented by the CSP RDD; RDD always reads a constant number of bytes from the input file (the number of bytes is the size of the base type of the file).

BTW: it is unclear to me at the moment, what exactly happens with variant records (same type, but varying length), if they were read in from and written out to files. There is still some work to be done (analyzing, testing, verifying, maybe fixing errors).

The sample program:

```
program TESTRDD ( INPFILE , OUTPUT ) ;

type BYTE = record
              CONT : CHAR
            end ;

var INPFILE : FILE of BYTE ;
    B : BYTE ;
    CH : CHAR ;

begin (* HAUPTPROGRAMM *)
  RESET ( INPFILE ) ;
  while not EOF ( INPFILE ) do
    begin
      READ ( INPFILE , B ) ;
      CH := B . CONT ;
      WRITE ( 'gelesen: ' ) ;
      if ORD ( CH ) > ORD ( ' ' ) then
        WRITE ( CH )
      else
        WRITE ( ' ' ) ;
      WRITE ( ORD ( CH ) : 4 ) ;
      CH := INPFILE -> . CONT ;
      WRITELN ( ORD ( CH ) : 4 ) ;
    end (* while *) ;
end (* HAUPTPROGRAMM *) .
```

Some more remarks:

- the type definition BYTE triggers the binary READ
- there are two characters written in the loop body: first the character returned by READ (= B.CONT) and then the next character (INPFILE →.CONT), which is the next character present in the file buffer
- RESET at the beginning puts the first character into the file buffer; READ assigns the file buffer to the output variable and then calls GET; that‘s why there is still a valid result after EOF in the READ result variable (in this case).

The result, if TESTRDD is given its own source code as input, will be like this (on Windows) – you can see the CR LF sequences:

```
gelesen: p 112 114
gelesen: r 114 111
gelesen: o 111 103
gelesen: g 103 114
gelesen: r 114  97
gelesen: a  97 109
gelesen: m 109  32
gelesen:    32  84
gelesen: T  84  69
gelesen: E  69  83
gelesen: S  83  84
gelesen: T  84  82
gelesen: R  82  68
gelesen: D  68  68
gelesen: D  68  32
gelesen:    32  40
gelesen: (  40  32
gelesen:    32  73
gelesen: I  73  78
gelesen: N  78  80
gelesen: P  80  70
gelesen: F  70  73
gelesen: I  73  76
gelesen: L  76  69
gelesen: E  69  32
gelesen:    32  44
gelesen: ,  44  32
gelesen:    32  79
gelesen: O  79  85
gelesen: U  85  84
gelesen: T  84  80
gelesen: P  80  85
gelesen: U  85  84
gelesen: T  84  32
gelesen:    32  41
gelesen: )  41  32
gelesen:    32  59
gelesen: ;  59  13
gelesen:    13  10
gelesen:    10  13
gelesen:    13  10

...
```

```
gelesen:    13  10
gelesen:    10 101
gelesen: e 101 110
gelesen: n 110 100
gelesen: d 100  32
gelesen:    32  40
gelesen: (  40  42
gelesen: *  42  32
gelesen:    32  72
gelesen: H  72  65
gelesen: A  65  85
gelesen: U  85  80
gelesen: P  80  84
gelesen: T  84  80
gelesen: P  80  82
gelesen: R  82  79
gelesen: O  79  71
gelesen: G  71  82
gelesen: R  82  65
gelesen: A  65  77
gelesen: M  77  77
gelesen: M  77  32
gelesen:    32  42
gelesen: *  42  41
gelesen: )  41  32
gelesen:    32  46
gelesen: .  46  13
gelesen:    13  10
gelesen:    10   0
```

I hope you enjoyed this story of New Stanford Pascal development;
please send comments and suggestions to

berndoppolzer@yahoo.com

or

bernd.oppolzer@t-online.de