Short language reference of New Stanford Pascal (04.2023)

First of all: please excuse possible errors in my English; I am German and not a native English speaker … I will do the best I can.

The New Stanford Pascal compiler implements the Pascal language as defined by the Jensen/Wirth publication "Pascal User Manual and Report" of the 1970s - but with many enhancements and some differences.

These enhancements are listed in this paper.

The language defined by Jensen/Wirth is called "Standard Pascal" in this paper.

1) Program header

As with Standard Pascal, the program header may contain a list of external file names. But the names of the external files are optional. Every file which is declared in the main function is external by default, no matter if it appears in the program header or not. The allocation of external Pascal files to files known to the operating system(s) is explained later.

Furthermore, there are six predefined external files, all of type TEXT:
INPUT, OUTPUT, PRR, QRR, PRD, QRD.

There is a second variant of program header, which starts with the MODULE keyword (instead of PROGRAM). This second variant does not contain a main program, but instead a collection of external procedures and functions (see later). A module must not have a list of file names, and the main routine must be empty in the module case (like this: BEGIN END).

A module must not have (global) variables, only procedures and functions. But: a module can have global STATIC variables (see later). These static variables can be initialized by constants (see later).

MODULE example (which contains some of the New Stanford Pascal enhancements – see later paragraphs):

```
module MLANGREF ;

const TAB_MAX = 1000 ;
      TAB_MAX2 : INTEGER = 1000 ;
      MESSAGE = 'This is a message' ;
      MESSAGE2 : CHAR ( 100 ) = 'This is a message' ;
      TAB : array [ 1 .. 5 ] of INTEGER =
      ( 10 , 20 , 30 , 40 , 50 ) ;

type DATE = record
              YEAR : 0 .. 9999 ;
              MONTH : 0 .. 12 ;
              DAY : 0 .. 31 ;
            end ;
     PERSON = record
                NAME : CHAR ( 30 ) ;
                BIRTH : DATE
              end ;

const X : PERSON =
      ( 'John' , ( 1957 , 7 , 4 ) ) ;

static CLONG : CHAR ( 32760 ) ;


procedure FILL_CLONG ( const X : CHAR ( 32760 ) ) ;

   begin (* FILL_CLONG *)
     MEMCPY ( ADDR ( CLONG ) , ADDR ( X ) , SIZEOF ( CLONG ) ) ;
   end (* FILL_CLONG *) ;


begin (* HAUPTPROGRAMM *)

end (* HAUPTPROGRAMM *) .
```

2) Constants

First of all, the "definition sections" CONST, TYPE, VAR etc., which must be in a certain order in Standard Pascal and which may appear only once at a certain block level, may appear in ANY order in New Stanford Pascal and may appear multiple times.

This is necessary, because we also have "typed" constants, that is: constant definitions involving explicit types, and so the type definitions in this case must precede the const definitions.

"Normal" consts look like in this example:

```
const TAB_MAX = 1000;
```

where the type of the symbolic constant (TAB_MAX in this case) is derived from the type of the constant (1000 in this case, which sets the type of TAB_MAX to integer).

With New Stanford Pascal, you can also specify consts like this:

```
const TAB_MAX2 : INTEGER = 1000;
```

which does not add much function.

But

```
const MESSAGE = 'This is a message';
```

is different from

```
const MESSAGE2 : CHAR (100) = 'This is a message';
```

the second form has length 100 and is padded with blanks to the right.

See more on the type CHAR (100) below; the type CHAR (100) is not present in Standard Pascal.

The type in such const definitions can be any type, for example:

```
const TAB : array [1..5] of INTEGER = (10, 20, 30, 40, 50);
```

Of course, you can use the complete type syntax after the colon, and you can use type identifiers of types defined before.

The so-called structured constants are specified using a new syntax (new = introduced in 1982 at McGill university). As a rule of thumb:

every record and every array has to be enclosed by parantheses in the constant specification.

See the following example for a record const.

```
type DATE = record
              YEAR : 0 .. 9999 ;
              MONTH : 0 .. 12 ;
              DAY : 0 .. 31 ;
            end ;
     PERSON = record
                NAME : CHAR ( 30 ) ;
                BIRTH : DATE
              end ;

const X : PERSON =
      ( 'John' , ( 1957 , 7 , 4 ) ) ;
```

3) Types

New Stanford Pascal supports all the types defined by Standard Pascal.

Sets

With the current implementation, the base type of sets is limited to 2000 elements. SET OF CHAR is possible, of course. SET OF 0 .. 1999 is possible, too.

At the moment there is a serious bug: the base type of a set must not contain negative integers ... this will be addressed in the near future.

Files

TEXT files are fully supported, with some significant extensions (see "Predefined Functions ...").

Binary files are supported, too, but only with constant length at the moment. Not sure what happens with binary files where the base type is a structure with variant records.

Records

New Stanford Pascal supports WITH clauses on sub records, see this example:

```
IDENTIFIER = record
               with BASE : IDENT_BASE ;
               KLASS : IDCLASS ;
               with PTYPE : -> IDENT_TYPE_EXT ;
               with PKONST : -> IDENT_KONST_EXT ;
               with PSTKONST : -> IDENT_STKONST_EXT ;
               with PVAR : -> IDENT_VAR_EXT ;
               with PFIELD : -> IDENT_FIELD_EXT ;
               with PPROC : -> IDENT_PROC_EXT ;
             end ;
```

the WITH clause has the effect that the components of the sub records are treated as components of the original record. This means, that in the example above, the selector BASE is optional, if a component of the BASE subrecord is selected from an IDENTIFIER record.

This also works, if the subrecord is not a real subrecord (as BASE), but accessed via pointer, as the other subrecords in the example above (PTYPE, PKONST etc.). This, of course, only works, if the pointer is set before accessing the subrecord.

In the example above, at any given time, only one of the subrecords PTYPE to PPROC is active (that is: the pointer is not NIL), depending on the value of KLASS.

PACKED

the PACKED keyword is accepted, but ignored. All char arrays are packed by default.

More types

There are some new types in New Stanford Pascal

- CHAR (n)

CHAR (n) is simply an abbreviation for PACKED ARRAY [1 .. n] OF CHAR. CHAR (n) - called CHAR arrays - are character strings which always have a fixed length. This type (but not the abbreviation) has existed since the time of Standard Pascal.

But: it is now possible to assign shorter CHAR array variables (or constants) to longer CHAR arrays (by a simple assignment). The longer CHAR array will be padded with blanks. This was NOT possible with Standard Pascal.

It is NOT possible to assign a longer CHAR array to a shorter one; you will have to use the SUBSTR function in this case.

CHAR variables can be defined with lengths up to 32 k.

CHAR array constants are limited to 254 bytes; they can simply be terminated on one line and continued on the next line.

- STRING (n)

In contrast to CHAR (n), STRING (n) is a varying length character string. A STRING variable has two control fields - one holding the maximum length as allowed by the definition, and the other holding the actual length (which may be zero). These fields can be queried by the MAXLENGTH and the LENGTH builtin functions.

It is possible to assign STRINGs to CHARs and vice versa. But keep in mind that, when assigning CHARs to STRINGs, the trailing blanks of the CHAR will be copied, too. To prevent this, you may use the RTRIM function (see later).

STRINGs can be concatenated etc. - see more on the later paragraphs on predefined functions etc.

In main storage, STRINGs are stored in their defined maximum length most of the time. During processing, the STRINGs may live in a so-called string workarea which is handled by the runtime system and which is garbage-collected automatically.

STRINGs can, of course, be part of arrays or records.

STRING variables can be defined with lengths up to 32 k.

- DECIMAL (n, m)

This type is supported and can be used. It will support decimal computations in the future. At the moment it is simply another name for the REAL type, with one exception:

if you WRITE a DECIMAL value, it is automatically written with the number of decimal digits as derived from the DECIMAL type definition.

See this example:

```
var X : DECIMAL (9, 2);
    ...
    WRITE (X);   /* same as WRITE (X : 11 : 2); */
```

n + 2 is used for the width specification, so that the decimal point and a minus (if needed) can be written, too.

With REAL instead of DECIMAL (9,2), WRITE (X) would write a representation with mantissa and exponent.

4) Variables

There is not much difference to Standard Pascal in the variable definitions with one exception:

New Stanford Pascal supports the initialization of variables at the time of the definition.

This is done using the constant syntax (even structured constants are allowed, see above).

for example:

```
var X : PERSON := ( 'John' , ( 1957 , 7 , 4 ) ) ;
    I : INTEGER := 0 ;
    OK : BOOLEAN := TRUE ;
```

the type person used here is defined above in the types section.

The only difference from the const definition above is the keyword var
and the := here instead of the = there.

In fact, the compiler accepts = and := at both places, but a warning is issued,
if := is used on const definitions and if = is used on var definitions.

The internal handling is different; the variables defined here live on the runtime stack and therefore the initializations must be carried out at runtime using machine instructions EVERY TIME when the block which contains these definitions is entered.

So such definitions can be costly, please keep this in mind.

5) Static Variables

Static variables can be specified much the same as "normal" (auto) variables by simply replacing the var keyword by the static keyword.

for example:

```
static X : PERSON := ( 'John' , ( 1957 , 7 , 4 ) ) ;
       I : INTEGER := 0 ;
       OK : BOOLEAN := TRUE ;
```

static is a new reserved word of New Stanford Pascal.

In contrast to variables with initialization, these initializations of static data are carried out once (at compile time), so that there is no performance penalty.

Modules (see above) cannot have global variables, but they can have global static data which are common to all (external) procedures and functions defined in a module. This is so-called static data local to the module (private storage hidden from the users of the interfaces of the module).

6) Procedures and Functions

a) Full type syntax in parameter lists

With Standard Pascal, in parameter lists for procedures and functions only type identifiers are allowed. This means that if you want to pass a pointer type (for example), you have to define this type explicitly and provide a type name for it.

Example:

```
type DATE = record
              YEAR : 0 .. 9999 ;
              MONTH : 0 .. 12 ;
              DAY : 0 .. 31 ;
            end ;
     PERSON = record
                NAME : CHAR ( 30 ) ;
                BIRTH : DATE
              end ;
     PPERSON = -> PERSON;
...
procedure CHECK_PERSON (P : PPERSON) : BOOLEAN ;
...
```

With New Stanford Pascal, it is possible to use the full type syntax on parameter lists, so that you don‘t need an explicit type definition in this case.

Example:

```
procedure CHECK_PERSON (P : -> PERSON) : BOOLEAN ;
```

This is also possible, if the result of a function is a pointer type:

```
function SEARCH_PERSON (X : CHAR (30)) : -> PERSON ;
```

b) CONST parameters

Standard Pascal, as defined in the Jensen/Wirth book, supports call by value (normal parameters) and call by reference (VAR parameters).

When passing parameters „by value“, you can transfer simple values or variables or complete expressions on the call; the expressions are evaluated and copied to the procedure or function (much like local variables), so that no change which the called function does to the parameters will be reflected to the caller‘s variables.

When passing parameter with VAR („by reference“), only variables can be transferred. In this case, a reference (the address) of the variable is passed to the caller, and the changes which the called function does to the parameter will be reflected to the caller‘s variable.

New Stanford Pascal support CONST parameters, too. With CONST, the caller can speficy expressions (much like „by value“), but technically a reference is passed to the caller (like in the VAR case). With simple types, the value is computed and copied into a compiler generated temporary field (a so-called dummy argument), then the address of this temporary field is passed. So it is guaranteed by this technique that no changes are reflected to the variables of the caller.

With complex types like large arrays or structures, there is no copy (like in the „by value“ case), but simply the address is passed like in the VAR case. But: the compiler in this case does (or should) not allow, that the parameters are changed in the called procedure or functions; this includes that they may not be passed to a lower function as VAR parameters.

It is important to notice that the parameters in the called functions are used all in the same way, no mattter if they are „normal“, VAR or CONST parameters ... there is no syntactic difference.

c) Conformant strings

When passing strings to procedures or functions, it works differently depending on the parameter passing mechanism;

- with call by value, you specify the length of the string on the parameter list. This is the length of the local copy of the parameter in the stack frame of the called function. The length of the passed argument may be shorter and will be padded to the length specified here during the function call. The passed argument may be a constant or a string expression.

- with call by reference (VAR parameter), you don‘t specify a length. That is, you only use the STRING type, no parantheses, no length. This is called a „conformant“ string. The control fields of the string (current length and maxlength) are derived from the value of the passed string and may be examined by the called function.

- same goes with a CONST parameter of type STRING; this is also a conformant string. The difference to VAR parameters is that a CONST parameter may not be changed by the called function. But technically it is the same. If you specify a string constant or a string expression for a CONST parameter of type string, it has to be evaluated during the function call and stored as temporary string somewhere in the string working area; the address of this temporary string is then passed.

BTW: there is another paper available from the New Stanford Pascal website, which covers all the details of String handling ... with many examples.

d) Visibility of procedures and functions

Procedures and functions which are defined in a Pascal main program are only known inside this program; they cannot by called from another program or module.

Procedures and functions defined in a module are visible from other programs or modules by default, as long as they are defined at the top level of the module. Procedures and functions defined at lower levels are not visible from outside.

You can hide procedures and functions (that is: make them invisible from outside) by adding the LOCAL keyword to the procedure definition.

Like this (see the MODULE example at the beginning of the document):

```
local procedure FILL_CLONG ( const X : CHAR ( 32760 ) ) ;

   begin (* FILL_CLONG *)
     MEMCPY ( ADDR ( CLONG ) , ADDR ( X ) , SIZEOF ( CLONG ) ) ;
   end (* FILL_CLONG *) ;
```

e) External procedures and functions in other languages

New Stanford Pascal supports calling external subroutines written in other languages; only on the mainframe at the moment, and only for FORTRAN and ASSEMBLER.

This is done by appending the language to the EXTERNAL keyword (e.g. EXTERNAL FORTRAN). If you want, you can add an 8 byte string which tells the external name of the procedure. If you don't, the external name is derived from the Pascal name.

See this example:

```
procedure PASCAL_TO_FORTRAN
          ( X1 : INTEGER ; var X2 : INTEGER ;
            T1 : CHAR20 ; var T2 : CHAR20 ) ;

   EXTERNAL FORTRAN 'PAS2FTN' ;

function PAS_TO_FTN_FUNC
         ( X1 : INTEGER ; X2 : INTEGER ) : INTEGER ;

   EXTERNAL FORTRAN 'PAS2FF' ;
```

BTW: there is another paper available from the New Stanford Pascal website, which covers all the details of external procedures, including parameter passing etc.

7) Statements

a) OTHERWISE clause on CASE statements

The CASE statement has an OTHERWISE clause which is executed, if no case label matches; see this example:

```
procedure ALIGN ( var Q : ADDRRANGE ; P : ADDRRANGE ) ;

   begin (* ALIGN *)
     case P of
       REALSIZE :
         Q := ( ( Q + 7 ) DIV 8 ) * 8 ;
       INTSIZE :
         Q := ( ( Q + 3 ) DIV 4 ) * 4 ;
       HINTSIZE :
         if ODD ( Q ) then
           Q := Q + 1 ;
       CHARSIZE :
         ;
       otherwise
         if SCB . FEZAHL = 0 then
           SET_ERROR ( 401 ) ;
     end (* case *) ;
   end (* ALIGN *) ;
```

b) CONTINUE and BREAK

The new CONTINUE statement can be used in WHILE, REPEAT and FOR loops;
it starts the next iteration of the loop.
The new BREAK statement can be used in WHILE, REPEAT and FOR loops;
it terminates the execution of the loop.

See this example; here the index SIX is incremented to access some elements in the char array ST until the end of the string (terminated by an apostroph) is found; if there are two apostrophs, the string does not yet end, because two apostrophs are used to encode an apostroph inside the string.

```
if INSTRING then
  while SIX <= LENGTH ( ST ) do
    begin
      if ST [ SIX ] <> '''' then
        begin
          SIX := SIX + 1 ;
          continue ;
        end (* then *) ;
      INSTRING := FALSE ;
      SIX := SIX + 1 ;
      if SIX > LENGTH ( ST ) then
        break ;
      if ST [ SIX ] = '''' then
        begin
          SIX := SIX + 1 ;
          INSTRING := TRUE ;
          continue ;
        end (* then *) ;
      break ;
    end (* while *) ;
```

c) RETURN

The new RETURN statement can be used in procedures or functions to leave the procedure or function. There is no value on the RETURN statement inside functions as in other languages, because the function result is returned by assigning a value to the function name.

See this (procedure) example:

```
procedure PARSE_PARM ( const ST : STRING ; var SIX : INTEGER ;
                       var PARM_IX : INTEGER ) ;

   begin (* PARSE_PARM *)
     PARM_IX := 0 ;
     if SIX > LENGTH ( ST ) then
       return ;
     while SIX <= LENGTH ( ST ) do
       begin
         if ST [ SIX ] <> ' ' then
           break ;
         SIX := SIX + 1 ;
       end (* while *) ;
     if SIX > LENGTH ( ST ) then
       return ;
     PARM_IX := SIX ;
   end (* PARSE_PARM *) ;
```

Note:

CONTINUE, BREAK and RETURN should eliminate most legitimate use cases of GOTOs from the past

8) Predefined Procedures and Functions

Theere are many new (and some old) predefined procedures and functions, which are not known in Standard Pascal. Some of the new functions are inspired by C, like MEMCPY, MEMSET, SIZEOF, ADDR.

Others are the string functions of Pascal/VS, like LENGTH, MAXLENGTH, SUBSTR, DELETE, INDEX, TRANSLATE etc.

PTRADD and PTRDIFF provide pointer arithmetic.

A new storage management (inspired by the LE runtime system) is available; see the procedures ALLOC and FREE.

This is the alphabetic list of ALL functions and procedures available in New Stanford Pascal at the moment (04.2023):

ABS	is a legacy Pascal function; it returns the absolute value of the argument
ADDR	new with New Stanford Pascal; ADDR returns the address of the argument (which should be a variable or an array element etc.); the value returned has the type ANYPTR (a type compatible to every other pointer type)
ALLOC	this new function allocates a number of bytes on the new LE inspired heap (much like C's malloc)
ALLOCX	directly calls the ALLOC function of the underlying OS (no management, no optimization)
APPEND	Opens a file for output. Used to append new content at the end of the file. Similar to REWRITE, but REWRITE starts from the beginning of the file and overwrites old content
ARCTAN	arcus tangens function (Fortran library on the mainframe)
CARD	legacy Pascal function
CHKALLOC	description will be added later
CHKHEAP	description will be added later
CHR	legacy Pascal function, converts integer to char
CLOCK	compute CPU time in clock units
CLOSE	new function: CLOSE a file
COMPRESS	replaces sequences of spaces in the string by only one space (like in Pascal/VS)
COS	cosinus function (Fortran library on the mainframe)
DELETE	deletes portions of a string; the parameters are the same as with SUBSTR. DELETE works the same as its Pascal/VS counterpart

DIGITSOF	returns the number of digits of a decimal variable (from the definition)
DISPOSE	legacy Pascal function, but has no effect in this implementation (!)
EOF	legacy Pascal function; returns TRUE if the end of the file (the function argument) is reached
EOL	special „Stanford“ flavor of EOLN
EOLN	legacy Pascal function; returns TRUE if the end of the input line is reached on the file, given as argument (during READ)
EOT	special „Stanford“ flavor of EOF
EXIT	this function terminates the program. The argument is the exit code, which is transferred to the operating system. (Some other Pascal variants have a similar procedure called HALT; some without exit code).
EXP	exponentation function (Fortran library on the mainframe)
EXPO	legacy „Stanford“ function; gets exponent of floating point number (implemented as P-Code instruction)
FILEFCB	returns the address of the FCB (file control block) of the given file
FLOOR	legacy Pascal standard math function; truncates the argument (floating point) to the next lower integer value.
FREE	frees the ALLOC area identified by the pointer argument
FREEX	same as FREE, but for areas which have been acquired using ALLOCX
GET	legacy Pascal standard proc; GET is used to position the file pointer to the next file element on input files.
INDEX	INDEX (s1, s2) returns the index of the first occurence of s2 in s1; zero if not found. Same as in Pascal/VS and PL/1.
LASTINDEX	LASTINDEX (s1, s2) returns the index of the last occurence of s2 in s1; zero if not found.
LEFT	string function. Same as SUBSTR starting from position 1.
LENGTH	gets the current length of a string variable
LINELIMIT	legacy Pascal standard proc; not sure, what this function does ...
LN	logarithm function (Fortran library on the mainframe)
LTRIM	trims spaces on the left (like in Pascal/VS)
MARK	legacy Pascal standard proc to save the current heap pointer, see RELEASE (MARK and RELEASE should not be used with new programs, but they are still used in the compiler; use ALLOC and FREE with new programs)
MAXLENGTH	gets the maximum length of a string variable

MEMCMP block compare like memcmp in C

MEMCPY block move (copy) like memcpy in C

MEMSET initialize a block with a byte pattern (like memset in C)

MESSAGE on the mainframe: WTO (write to operator);
on other platforms: write to stderr

NEW legacy Pascal standard proc to allocate storage on the (old) heap;
such storage can only be freed by using MARK and RELEASE.
It is recommended to use ALLOC and FREE with new programs
instead of NEW.

ODD legacy Pascal standard proc, returns TRUE, if the argument is odd,
and FALSE otherwise

ORD legacy Pascal standard proc; converts char to integer

PACK legacy Pascal standard proc - only for compatibility reasons,
not needed

PAGE legacy Pascal standard proc; writes a new page on an output file

PRECISIONOF returns the precision of a decimal variable (from the definition);
that is, the number of digits after the decimal point

PRED legacy Pascal standard proc; returns the predecessor of the argument
(with scalar and simple types)

PTR2INT new function; converts pointers to integers

PTRADD new function; adds an integer value to a pointer

PTRCAST new function; PTRCAST can be used to assign pointers
of different types

PTRDIFF new function; PTRDIFF returns the difference of two pointers
as an integer value

PUT legacy Pascal standard proc; used to transfer a file component to a file
(usually not used for TEXT files, but for binary files, that is:
FILES of RECORDs).

READ legacy Pascal standard proc; used to read different types from a TEXT
file. New Stanford Pascal has several extensions; it is possible to read
scalar types, for example. And: READ supports length specifications,
like with Pascal/VS. For example:

```
READ (I:4, CH, J:5);
```

only 4 characters are read for the integer value I,
then one character CH, then 5 characters for J etc.

READLN legacy Pascal standard proc; used to ignore the rest of the current line and start reading at the beginning of the next line

READSTR new function, same as in Pascal/VS. The first parameter is a string variable. The variables are read from the string variable and not from an input file.

Example:

```
S := '0007X   12';
READSTR (S, I:4, CH, J:5);
```

This will set I to 7, CH to 'X' and J to 12.

RELEASE legacy Pascal standard proc to reset the heap pointer; this way returning all storage since the last MARK call to the Pascal runtime (should not be used with new programs; use ALLOC and FREE instead).

REPEATSTR REPEATSTR (s, n) returns the string s repeated n times. This function is implemented using a new P-Code instruction, and it is used inside string functions to build new strings, containing of n blanks (for example)

RESET legacy Pascal standard proc; this procedure opens a file for input and resets the file pointer to the beginning of the file

RESULTP is a function returning a pointer (ANYPTR) to the function result (usable in every function)

REWRITE legacy Pascal standard proc; this procedure opens a file for output and resets the file pointer to the beginning of the file. Old content of the file is lost

RIGHT string function. Similar to SUBSTR at the right part of the string.

ROUND legacy Pascal standard proc; returns the argument (floating point) rounded as integer value. Caution: if the FP argument is the result of a computation, you may not get the expected result due to precision problems.

ROUNDX new function with two arguments. The first argument is a FP value, and the second tells the decimal place, where the rounding will take place. For example: roundx (123.4567, 2) = 123.46

The result is a FP value again. The input FP value is "corrected" before the rounding takes place, so that precision problems (hopefully) will not occur.

RTRIM trims spaces on the right (like TRIM in Pascal/VS)

SIN sinus function (Fortran library on the mainframe)

SIZEOF this (compile time) function returns the size of an object in bytes; it can be used for variables and for types (much the same as in C)

SKIP	legacy Pascal standard proc; writes line feeds on output files (maybe!)
SQR	legacy Pascal standard proc ... maybe square ?
SQRT	square root function (Fortran library on the mainframe)
STR	this function converts char arrays and single chars to strings
STRRESULT	is a function of type STRING which retrieves the current function result, when inside a string function
STRRESULTP	is a function of type pointer to CHAR, which points to the content of the current function result, when inside a string function
SUBSTR	SUBSTR (s, n, m) or SUBSTR (s, n) - gets the Substring from s starting at position n with length m (first format) or the rest of the string (2nd format); same as in Pascal/VS and PL/1
SUCC	legacy Pascal standard proc; returns the successor of the argument (with scalar and simple types)
TRACE	legacy Pascal standard proc; not sure, what this function does ...
TRANSLATE	TRANSLATE (s1, to, from) translates s1 controlled by one or two translate tables (see PL/1)
TRAP	legacy Pascal standard proc; not sure, what this function does ...
TRIM	trims spaces on both sides (different from TRIM in Pascal/VS !!)
TRUNC	legacy Pascal standard proc; returns the argument (floating point) truncated to the next lower integer value.
UNPACK	legacy Pascal standard proc - only for compatibility reasons, not needed
VERIFY	VERIFY (s1, s2) returns the index of the first character in s1 which is not in s2 (same as in PL/1)
WRITE	legacy Pascal standard proc; used to write different types to an output file. New Stanford Pascal has several extensions; it is possible to write scalar types and pointers, for example.
WRITELN	legacy Pascal standard proc; writes a new line on output files
WRITESTR	new function, same as in Pascal/VS. The first parameter is a string variable. The result of the WRITE goes to the string variable and not to a file. The same types and parameters are allowed as with "normal" WRITE; the string must be long enough to hold the result.

9) Allocation of external (file system) files to Pascal files

On mainframe systems, the names of the Pascal external files are simply used as DDNames and can be assigned to "real" files using the mechanisms of the operating system used (DD statements on MVS or z/OS, FILEDEF on VM/CMS).

The name of the Pascal external file, as seen from outside, is the name of the Pascal file variable, but only the first eight characters. All files which are defined in the main procedure (top level), are external, no matter if they appear in the program statement or not.

Because modules don't have a top level procedure (and no top-level variables), they don't have external file variables of their own.

But:

- (external) procedures or functions within modules can access files passed to them by (var) parameter

- they can access the six predefined files

- modules can have a pointer to a file in the global (private) static area (which must be set by a procedure call, before it can be used)

- at the moment, a module cannot have a file declaration in its static area

On non-mainframe systems, the allocation of files works virtually the same way as on mainframes.

The Pascal external file names (first 8 bytes only) are used to access the file system, if nothing special is done.

But: you can override this filename by using an environment variable.

On Windows:

```
SET DD_INPFILE=c:\work\inpfile.txt
```

this sets the "real" filename for the Pascal file INPFILE.

```
SET DD_INPFILE=
```

this resets the environment; the "real" file INPFILE is used

```
SET DD_OUTFILE=*stdout*
```

all Pascal writes to file OUTFILE go to stdout.

The file INPUT is assigned to stdin by default (but can be reassigned);
the file OUTPUT is assigned to stdout by default (but can be reassigned).

10) Compiler Options

Compiler options can be specified by a special comment syntax.

BTW: comments in the source code can have several formats:

```
/* PL1 and C style */
(* Pascal style 1 *)
{ Pascal style 2 }
// C++ style - until end of line
```

Comments of the same format can be nested, if the compiler option N is set (which is on by default). With the compiler option N set, this is a valid comment:

```
(* nested comment (* Pascal style *) *)
```

A compiler option is a comment, which starts with a $ (dollar character).
It works with all 4 comment formats.

For example:

```
//$A+,D-,N+
```

A+ activates the ASSEMBLER listing (disabled by default), and D- disables the debugging features (which are enabled by default). N+ sets the N option (for the nesting of comments).

Same effect with a Pascal style comment:

```
(*$A+,D-,N+ ... various options set *)
```

A complete list of compile options will be provided later.

11) Run Time Options

Run Time Options to control some parameters of the runtime systems can be specified at program start; details will be provided later.

12) Command Line Parameters

A (main) program can access the command line parameters by a predefined variable call OSPARM.

OSPARM is a pointer which points to a record; the record contains
- an integer PLENGTH (length of the parameter string)
- and a char array PSTRING [1 .. PLENGTH] of CHAR.

OSPARM may be NIL; in this case no parameter string was given.

The following code snippet extracts a SOURCENAME value from the parameter string (if present):

```
var SOURCENAME: CHAR (8);
    SX : INTEGER;

...

if OSPARM = NIL then
  SOURCENAME := ''
else
  with OSPARM -> do
    begin
      SOURCENAME := '' ;
      for SX := 1 to 8 do
        if SX <= PLENGTH then
          SOURCENAME [ SX ] := PSTRING [ SX ] ;
    end (* with *) ;
```

I hope you like this short language reference of New Stanford Pascal;
please send comments and suggestions to

berndoppolzer@yahoo.com

or

bernd.oppolzer@t-online.de