AsciiMappers

```
DEFINITION LibAsciiMappers;

   IMPORT Files, TextModels;

   CONST
      commaSep = 3;
      freeFormat = 11;
      invalid = 31;
      isBool = 6;
      isEof = 30;
      isEol = 20;
      isInt = 4;
      isLInt = 16;
      isReal = 5;
      isSet = 7;
      isStr = 3;
      isSym = 18;
      lcExponent = 10;
      seekBool = 6;
      seekSet = 9;
      seekSym = 7;
      spcSep = 2;
      stopIsSym = 8;
      tabIsSpace = 0;
      tabSep = 4;
      trimSpc = 1;
      typeTrap = 12;
      useQuotes = 5;

   TYPE
      Str = POINTER TO ARRAY OF CHAR;

      Scanner = EXTENSIBLE RECORD
         type-, int-, lineNo-, tknNo-, begCol-, noCols-: INTEGER;
         lInt-: LONGINT;
         bool-: BOOLEAN;
         opts-, set-: SET;
         real-: REAL;
         sym-: CHAR;
         line-, token-: Str;
         typeErr: BOOLEAN;
         (VAR sc: Scanner) BaseFile (): Files.File, NEW;
         (VAR sc: Scanner) BaseText (): TextModels.Model, NEW;
         (VAR sc: Scanner) Bool (): BOOLEAN, NEW;
         (VAR sc: Scanner) Column (): INTEGER, NEW;
         (VAR sc: Scanner) ConnectTo (pntr: ANYPTR; opts: SET), NEW;
         (VAR sc: Scanner) ConnectToDisc (IN relPath, fileName: ARRAY OF CHAR; opts: SET), NEW;
         (VAR sc: Scanner) ConnectToFile (file: Files.File; opts: SET), NEW;
         (VAR sc: Scanner) ConnectToText (text: TextModels.Model; opts: SET), NEW;
         (VAR sc: Scanner) Int (): INTEGER, NEW;
         (VAR sc: Scanner) LInt (): LONGINT, NEW;
         (VAR sc: Scanner) Mark, NEW;
         (VAR sc: Scanner) Pos (): INTEGER, NEW;
         (VAR sc: Scanner) PreScan-, NEW, EMPTY;
         (VAR sc: Scanner) ReadLine, NEW;
         (VAR sc: Scanner) Real (): REAL, NEW;
         (VAR sc: Scanner) Rewind, NEW;
         (VAR sc: Scanner) Scan, NEW;
         (VAR sc: Scanner) Set (): SET, NEW;
```

```
        (VAR sc: Scanner) SetColumn (col: INTEGER), NEW;
        (VAR sc: Scanner) SetOpts (opts: SET), NEW;
        (VAR sc: Scanner) SetPos (pos: INTEGER), NEW;
        (VAR sc: Scanner) String (): Str, NEW;
        (VAR sc: Scanner) Symbol (): CHAR, NEW
      END;

END LibAsciiMappers.
```

An *AsciiMappers.Scanner* is a Mapper that uses a Files.Reader to scan structured ASCII text files, or a TextModels.Reader to scan structured ASCII Texts.
(Characters beyond 255 are generally SHORTened, and Views are projected. The one exception is that the Unicode '–' character, 2212X, is replaced by the ASCII '-' character 2DX.)

The standard import abbreviation for this module is '`IMPORT  Asc := LibAsciiMappers;`'.

AsciiMappers.Scanner is considerably more efficient than the (incomplete) approach outlined in ObxAscii for reading large text files, and offers greater flexibility. In particular, it offers the ability to read an unparsed line and allow the user to parse this using column number information, as well as offering more 'free format' options.

For simplicity and efficiency the Scanner can only make context free interpretations of scanned tokens; it is strictly limited to 1 character look-ahead.

A requirement, for example, to recognise a string such as:
   "35 + 5.9i" as a complex number, but
   "35 +Dog" as an integer followed by a string
is beyond the scope of the Scanner which will have already decided that "35" was an integer by the time it read the "+".
This kind of functionality needs to be implemented externally at a higher layer. To assist with such context aware interpretations the Scanner provides the methods *Mark* and *Rewind*.

The Scanner operation is a 4-stage process:

1 (Optional) Tabs (09X) in the input file are translated into spaces. (This stage requires the option bit *tabIsSpace* to be set.

2 'Token's are scanned from the input file.
Tokens are stretches of text between separators.
Separators comprise 1 or more contiguous characters comprising (in any order):
    0 or more spaces (if option bit *spcSep* is set - which it is by default)
    0 or more tabs (if option bit *tabSep* is set)
    0 or 1 commas (if option bit *commaSep* is set)
Carriage returns and linefeeds also count as separators.

3 (Optional) Spaces are trimmed from the beginning and end of each token. (This stage requires the option bit *trimSpc* to be set.
→ Option is irrelevant if spaces are being interpreted as separators.
→ Option can be important for reading numbers because:
        INTEGER & LONGINT tokens may not contain spaces
        REAL tokens may not contain trailing spaces.

4 Tokens are interpreted as INTEGERs, LONGINTs, REALs, BOOLEANs, SETs, Symbols, or Strings. (Note that strings can be of zero length.)

There is currently no 'Formatter' specialised to ASCII text files; use a LibFmtrs.Fmtr or a TextMappers.Formatter to write to a TextModels.Model (a Text), then externalise this complete as ASCII using Converters.Export (with HostTextConv.ExportText or another suitable Converter). (Note that the TextModels.Model needs to be wrapped in a TextViews.View for most Converters.)

Application guidelines:

The simplest way to read an unstructured ASCII file is indicated in the outline below:

```
VAR
  file  :  Files.File;
  sc    :  Asc.Scanner;
BEGIN
  file  :=  ...;
  sc.ConnectToFile (file, {Asc.freeFormat});

  sc.Scan;
  WHILE  sc.type  #  Asc.isEof  DO
    CASE  sc.type  OF
      Asc.isInt   :  Consume sc.int ...
  |   Asc.isLInt  :  Consume sc.lInt ...
  |   Asc.isReal  :  Consume sc.real ...
  |   Asc.isBool  :  Consume sc.bool ...
  |   Asc.isSym   :  Consume sc.symb ...
  |   Asc.isSet   :  Consume sc.set ...
  |   Asc.isStr   :  Consume sc.token ...
    END;
    sc.Scan
  END;
  sc.ConnectTo (NIL, {})
END
```

If the file structure is line based the following outline may be suitable:

```
BEGIN
  file  :=  ...;
  sc.ConnectToFile (file, {});

  WHILE  sc.type  #  Asc.isEof  DO
    sc.Scan;
    WHILE  sc.type  #  Asc.isEol  DO
      CASE  sc.type  OF
        Asc.isInt   :  Consume sc.int ...
  |     Asc.isLInt  :  Consume sc.lInt ...
  |     Asc.isReal  :  Consume sc.real ...
  |     Asc.isBool  :  Consume sc.bool ...
  |     Asc.isSym   :  Consume sc.symb ...
  |     Asc.isSet   :  Consume sc.set ...
  |     Asc.isStr   :  Consume sc.token ...
      END;
      sc.Scan
    END;
    sc.ReadLine
  END;
  sc.ConnectTo (NIL, {})
END
```

Note 1: In the first example the Scan loop exit condition uses `Asc.isEof`, whereas in the second example it uses `Asc.isEol` (the ReadLine loop exit condition uses `Asc.isEof`).

Note 2: The Scan loop requires an initial *sc.Scan* prior to the loop. The ReadLine loop does not, because *sc.ConnectToFile* (or *sc.ConnectToText*) includes the initial *sc.ReadLine* for you.

It would apparently be possible to eliminate the initial *sc.Scan* also, by including it in *sc.ReadLine*. This has not been done because for more structured ASCII files it may be desired to not use *sc.Scan*. Alternatives are to use the other routines (eg *sc.Real*), or to parse the line directly (eg by simply skipping over lines known to be irrelevant to the task in hand).

For more structured files the code may be matched to the known structure. A highly structured example might be (eg ten lines of four comma-separated numbers, followed by some optional comments):

```
  file  :=  ...;
```

```
    sc.ConnectToFile (file, {Asc.commaSep})

    NEW (mat, 10, 4);
    FOR  row  :=  0  T0  9  DO
      FOR  col  :=  0  TO  3  DO
        mat [row, col]  :=  sc.Real ()
      END;
      sc.ReadLine
    END;
    sc.ConnectTo (NIL, {})
    ASSERT (~sc.typeErr, 45)
```

When planning an application that will use an Ascii Scanner it is worth spending some time to consider the most appropriate set of options to use, remembering that it can be convenient to change the set of valid options midway through reading the File or Text.

Reference Manual:

CONST **tabIsSpace**
Option element.
Possible element of *Scanner.opts*.
If present, the scanner will convert tab characters (09X) to space characters when reading the file.

CONST **trimSpc**
Option element.
Possible element of *Scanner.opts*.
If present, the scanner will trim spaces from the beginning and end of each 'token'.

CONST **spcSep**
Option element. (Space separator.)
Possible element of *Scanner.opts*.
If present, the scanner regards spaces ( ) as white space, so they can be used as delimiters, and they will be scanned past.
Note: This option is set by default unless *commaSep* or *tabSep* is set.

CONST **commaSep**
Option element. (Comma separator.)
Possible element of *Scanner.opts*.
If present, the scanner will treat commas (,) as white space, so they can be used as delimiters, and they will be scanned past.
Note: If *commaSep* is set then *spcSep* is not set by default.

CONST **tabSep** (Tab separator.)
Option element.
Possible element of *Scanner.opts*.
If present, the scanner will treat tab characters (09X) as delimiters, and they will be scanned past.
If Scanner.opts contains *tabIsSpace* then *tabSep* becomes irrelevant as there will be no tabs in the read text.
Note: If *tabSep* is set then *spcSep* is not set by default.

CONST **useQuotes**
Option element
Possible element of *Scanner.opts*.
Uses quotation marks to identify strings.
For example the text sequence <'The quick brown fox'> would normally scan as four strings:
<'The>, <quick>, <brown>, & <fox'>.
With *useQuotes* in *Scanner.opts* it becomes one string:

<The quick brown fox>.

Either ' or " can be used, but they must form a matched pair. The other can be embedded in the quoted string. Quoted strings are automatically terminated at the end of the current text line.

CONST **seekBool**
Option element.
Possible element of *Scanner.opts*.
If present, the scanner will regard the strings 'TRUE' & 'FALSE' as BOOLEAN values. Otherwise they are simply treated as strings.

CONST **seekSym**
Option element.
Possible element of *Scanner.opts*.
If present, the scanner will regard punctuation marks as isolated tokens, called symbols. Otherwise they are simply treated as (parts of) strings.

The characters regarded as symbols are (in HEX):

`9:`	Tab
`21 .. 2A:`	! " # $ % & ' ( ) *
`2C:`	,
`2F:`	/
`3A .. 40:`	: ; < = > ? @
`5B .. 5E:`	[\ ] ^
`7B .. 7E:`	{ | } ~
`0A0:`	Nbs
`0A2 .. 0AF:`	¢ £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯
`0B0 .. 0BF:`	° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿
`0D7:`	×
`0F7:`	÷

[Nbs is the Non-breaking space.]

Notable omissions are '+', '-', '.', & '_', but see *stopIsSym* below.
The first three characters can form valid parts of numbers. Regarding them as symbols could fragment well-formed numbers, which are regarded as more basic constituents of a typical file. Underscore is treated as a normal letter.

CONST **stopIsSym**
Option element.
When set the characters '+', '-', & '.' are regarded as symbols (see *seekSym* above).
Has no effect if *seekSym* is not included.

CONST **seekSet**
Option element.
Possible element of *Scanner.opts*.
SETs can be represented in a format like {}, or {2, 3, 6 .. 8, 12}.
SETs cannot be scanned if *Scanner.opts* contains *seekSym*.

CONST **lcExponent**
Option element. (Lower case exponent.)
Possible element of *Scanner.opts*.
Normally REAL numbers expressed in exponential format use 'D' or 'E' as the exponent marker. With *lcExponent* set the characters 'e' & 'd' are also accepted.

CONST **freeFormat**
Option element
Possible element of *Scanner.opts*.
Normally the scanner reads a file line by line (by calling Scanner.ReadLine), then scans each line (for example by calling Scanner.Scan).
If *freeFormat* is set it is not necessary to call Scanner.ReadLine, and Scanner.Scan will read the entire file ignoring line ends.

CONST **typeTrap**
Option element
Possible element of *Scanner.opts*.
If one of the routines *Scanner.Int*, *Scanner.LInt*, *Scanner.Real*, *Scanner.Bool* or *Scanner.Set* finds an unsuitable token next in the file it normally simply sets the flags *Scanner.type* to be invalid and *Scanner.typeErr* to be TRUE.

If *typeTrap* is set this situation will cause a run-time TRAP.

Not finding a token (eg End-of-Line or End-of-File) does NOT count as finding an unsuitable token.

The following constant values are based on the corresponding constants defined in TextMappers:

CONST **isStr**
Possible value of *Scanner.type*.
Indicates that the last token read was interpreted as a string, and its value is in *Scanner.token*.

CONST **isInt**
Possible value of *Scanner.type*.
Indicates that the last token read was interpreted as an INTEGER, and its value is in *Scanner.int*.

CONST **isLInt**
Possible value of *Scanner.type*.
Indicates that the last token read was interpreted as a LONGINT, and its value is in *Scanner.lInt*.

CONST **isReal**
Possible value of *Scanner.type*.
Indicates that the last token read was interpreted as a REAL, and its value is in *Scanner.real*.

CONST **isBool**
Possible value of *Scanner.type*.
Indicates that the last token read was interpreted as a BOOLEAN, and its value is in *Scanner.bool*.

CONST **isSet**
Possible value of *Scanner.type*.
Indicates that the last token read was interpreted as a SET, and its value is in *Scanner.set*.

CONST **isSym**
Possible value of *Scanner.type*.
Indicates that the last token read was interpreted as a punctuation CHARacter, and its value is in *Scanner.sym*.

CONST **isEol**
Possible value of *Scanner.type*.
Indicates that the last Scan found no token, but hit the end of the current line.
Only set if *freeFormat* is not in *Scanner.opts*.

CONST **isEof**
Possible value of *Scanner.type*.
Indicates that Scanner.ReadLine found no more lines, but hit the end of the file.
Also (if *freeFormat* is in *Scanner.opts*) indicates that the last Scan found no token, but hit the end of the file.

CONST **invalid**
Possible value of *Scanner.type*.
Indicates that the last Scan failed to find the requested type, or that there have been no Scans since the Scanner was connected to the file.

TYPE **Str**
String type used by *Scanner.line*, *Scanner.token*, and *Scanner.String*.

TYPE **Scanner**

A Scanner is an object used to scan structured ASCII text files. It searches through an ASCII file skipping over 'White space'; typically spaces (20X), tabs (09X), carriage returns (0DX), linefeeds (0AX), and control characters (characters before space in the ASCII character set).

When it finds non-white characters it (generally) collects the largest contiguous section together, and this stretch of non-white characters is called a 'token'. The nature (type) of the token is then ascertained (eg one of INTEGER, LONGINT, REAL, BOOLEAN, Symbol, or String), and the token, its type, and its 'value' are recorded in the Scanner.

type-: INTEGER
The type of the last scanned token.
type will be one of:

isStr, *isInt*, *isReal*, *isBool*, *isSet*, *isLInt*, *isSym*, *isEol*, *isEof*, or *invalid*.

int-: INTEGER
If *type* = *isInt*, *int* is the value of the last token scanned.

lineNo-, **tknNo**-: INTEGER
lineNo identifies the line in the ASCII file currently being scanned.
lineNo starts at 0 for the first line.
lineNo is initialised to -1 by Scanner.ConnectToFile or Scanner.ConnectToText.

tknNo is the index number (count) of the token last scanned.
tknNo starts at 0
tknNo is initialised to -1 by Scanner.ConnectToFile or Scanner.ConnectToText.
tknNo is also initialised to -1 by Scanner.ReadLine *provided that freeFormat is NOT in Scanner.opts*.

begCol-, **noCols**-: INTEGER
The column numbers describing the position of the last scanned token in the current line.

lInt-: LONGINT
If *type* = *isLInt*, *lInt* is the value of the last token scanned.

bool-: BOOLEAN
If *type* = *isBool*, *bool* is the value of the last token scanned.

opts-: SET

The set of options currently being used to control the operation of the Scanner.
The supported options are:

tabIsSpace, *trimSpc*, *spcSep*, *commaSep*, *tabSep*, *useQuotes*, *seekBool*, *seekSet*, *seekSym*, *stopIsSpc*, *seekSet*, *lcExponent*, *freeFormat*, & *typeTrap*.

set-: SET
If *type* = *isSet*, *set* is the value of the last token scanned.

real-: REAL
If type = *isReal*, real is the value of the last token scanned.

sym-: CHAR
If *type* = *isSym*, *sym* is the value of the last token scanned.

line-: Str
The last line of text read from the ASCII file, and currently being scanned.
Note that LEN (line) = LEN (line$) + 1. ie the line is just the right length to hold the data and the trailing 0X.
line is immutable; its contents are never changed by the Scanner. Hence client code can take a copy by simply copying the pointer *sc.line*.
When a new line is read a new ARRAY is created, and the pointer *sc.line* is updated.

It is quite possible for application code to call sc.ReadLine, then to pre-process *sc.line*, then to use sc.Scan to parse the line. The extension hook *sc.PreScan* is provided for this purpose.

token-: Str
The last token read from the currently being scanned line.
Note that LEN (token) = 256, but that LEN (token$) can be anything in the range 0 .. 255. *sc.token* (the pointer) is never changed (except by sc.ConnectToFile or sc.ConnectToText), but its contents are updated every time *Scanner.Scan* (or one of *sc.Int* etc) is called. There is a maximum token length limitation of 255 characters.

typeErr: BOOLEAN
typeErr is set TRUE if one of the routines *Int*, *LInt*, *Real*, *Bool*, *Set*, or *String* finds a token that cannot be 'valued' in the requested type.

For example '1234' is an acceptable token for *Int*, *LInt*, *Real* or *String*.
Again '1234.' is an acceptable token for *Real* or *String*, not for *Int* or *LInt*.
Also 'TRUE' is only an acceptable token for *Bool* if *seekBool* is in *Scanner.opts*.
(Incidently anything is always acceptable for *String*.)

typeErr is set FALSE by a *Scanner.ConnectToFile* or *Scanner.ConnectToText*.
typeErr is never otherwise set FALSE by a Scanner. Hence it can be used to determine if the last series of scans were all acceptable. It may be set FALSE by application code.

PROCEDURE (VAR sc: Scanner) **BaseFile** (): Files.File
Returns the File that the Scanner is currently connected to, if any.

PROCEDURE (VAR sc: Scanner) **BaseText** (): TextModels.Model
Returns the Text that the Scanner is currently connected to, if any.

PROCEDURE (VAR sc: Scanner) **Bool** (): BOOLEAN
Scans for the next token, and tries to interpret it as a Boolean value.
(Note that the recognised strings are 'TRUE' & 'FALSE'; spaces are not permitted.)

If the token is not acceptable (which requires *sc.opts* to contain *seekBool*) *sc.type* is set to *invalid*.
In addition, if *sc.opts* contains *typeTrap*, the programme traps (TRAP 85).

If *sc.type* # *isBool* *sc.typeErr* is set TRUE.
Note that this includes the cases *sc.type* = *isEol* & *sc.type* = *isEof*, which do not cause a Trap.

PROCEDURE (VAR sc: Scanner) **Column** (): INTEGER
Identifies the position of the Scanner in the current line.

PROCEDURE (VAR sc: Scanner) **ConnectTo** (pntr: ANYPTR; opts: SET)
If *pntr* is NIL, or of type Files.File, ConnectTo is equivalent to ConnectToFile - see below.
If *pntr* is NIL, or of type TextModels.Model, ConnectTo is equivalent to ConnectToText - see below.

The scanner's options are set to *opts* - see SetOpts below - unless *pntr* = NIL in which case *opts* is ignored.

Pre
20 *pntr* is NIL, or of type Files.File, or of type TextModels.Model

PROCEDURE (VAR sc: Scanner) **ConnectToDisc** (IN relPath, fileName: ARRAY OF CHAR; opts: SET)
This is an 'omnibus' procedure that opens a file (specified by its relative path *relPath* & *fileName*), connects the Scanner to it (see *ConnectToFile* below), then sets its options to *opts*.
fileName should include the file's extension.

Pre
40 Path not found
50 File not found

PROCEDURE (VAR sc: Scanner) **ConnectToFile** (file: Files.File; opts: SET)
Disconnect the scanner from the file or text it was connected to previously (if any), closes the previous file (if any), and connects the scanner to the given file (if any).

The scanner's options are set to *opts* - see SetOpts below - unless *file* = NIL in which case *opts* is ignored.

When you have finished with a file call sc.ConnectToFile (NIL, {}). This frees up any resources used. For example, if the last line of the file was 10 MByte then *sc.line* would be 20 MByte cluttering up your heap. It also closes the file.

Note that sc.ConnectToFile (NIL, opts) is totally equivalent to sc.ConnectToText (NIL, opts).

PROCEDURE (VAR sc: Scanner) **ConnectToText** (text: TextModels.Model; opts: SET)
Disconnect the scanner from the file or text it was connected to previously (if any), closes the previous file (if any), and connects the scanner to the given text (if any).

The scanner's options are set to *opts* - see SetOpts below - unless *text* = NIL in which case *opts* is ignored.

When you have finished with a text call sc.ConnectToText (NIL, {}). This frees up any resources used. For example, if the last line of the text was 10 MByte then *sc.line* would be 10 MByte cluttering up your heap.

Note that sc.ConnectToText (NIL, opts) is totally equivalent to sc.ConnectToFile (NIL, opts).

PROCEDURE (VAR sc: Scanner) **Int** (): INTEGER
Scans for the next token, and tries to interpret it as an INTEGER value.
(Note that INTEGERs may contain neither leading nor trailing spaces.)

If the token is not acceptable *sc.type* is set to *invalid*.
In addition, if *sc.opts* contains *typeTrap*, the programme traps (TRAP 82).

If *sc.type # isInt sc.typeErr* is set TRUE.
Note that this includes the cases *sc.type* = *isEol* & *sc.type* = *isEof*, which do not cause a Trap.

PROCEDURE (VAR sc: Scanner) **LInt** (): LONGINT
Scans for the next token, and tries to interpret it as a LONGINT value.
(Note that LONGINTs may contain neither leading nor trailing spaces.)

If the token is not acceptable *sc.type* is set to *invalid*.
In addition, if *sc.opts* contains *typeTrap*, the programme traps (TRAP 83).

If *sc.type # isLInt sc.typeErr* is set TRUE.
Note that this includes the cases *sc.type* = *isEol* & *sc.type* = *isEof*, which do not cause a Trap.

PROCEDURE (VAR sc: Scanner) **Mark**
Marks, internally to the Scanner, the current position of the Scanner in the ASCII file.
Mark is automatically called by *sc.ConnectToFile* or *sc.ConnectToText*.

PROCEDURE (VAR sc: Scanner) **Pos** (): INTEGER
Returns the 'position' of the scanner in the File/Text.
Note that *sc.pos* is the position of the start of the line currently being scanned, and *sc.begPos* is the position of the start of the last token scanned within this line. Pos returns *sc.pos + sc.begPos*.

PROCEDURE (VAR sc: Scanner) **PreScan**-, EMPTY
Called by *sc.ReadLine*.
A hook procedure that can be implemented to pre-process the read line prior to calling *sc.Scan*, or the other type specific Scan routines (eg *sc.Real*).

An example might be if the file uses ';'s as separators. The pre-scan could change the ';'s into ','s, which can then be interpreted as separators by *sc.Scan*.

Another example might be that it is known that anything to the right of column 20 is a 'comment' to be ignored by the application. The simple assignment sc.line [20] := 0X will stop the Scanner looking there.

Pre
  sc.line # NIL

PROCEDURE (VAR sc: Scanner) **ReadLine**
Reads the next line from the ASCII file, and sets the Scanner to the start of the line.

If the last line has already been read *sc.type* is set to *isEof* and *sc.line* is set to NIL.
The first line is automatically read by *sc.ConnectToFile* or *sc.ConnectToText*.

If *freeFormat* is in *sc.opts* there is no need to ever call *ReadLine*, but it may be called to terminate scanning the current line.

The end-of-line markers accepted are CR (carriage return = 0DX), LF (line feed = 0AX), or CR immediately followed by LF.
(LF immediately followed by CR is interpreted as two end-of-line markers.)

ReadLine calls *sc.PreScan* (if a line is found).

PROCEDURE (VAR sc: Scanner) **Real** (): REAL
Scans for the next token, and tries to interpret it as a REAL value.
(Note that REALs may contain leading spaces, but not trailing spaces.)

If the token is not acceptable *sc.type* is set to *invalid*.
In addition, if *sc.opts* contains *typeTrap*, the programme traps (TRAP 84).

If *sc.type* # *isReal sc.typeErr* is set TRUE.
Note that this includes the cases *sc.type* = *isEol* & *sc.type* = *isEof*, which do not cause a Trap.

PROCEDURE (VAR sc: Scanner) **Rewind**
Resets the position of the Scanner to where it was when sc.Mark was last called.

Note: the full state of the Scanner is not reset. *sc.type* is set to *invalid*. It is updated after the next Scan.

PROCEDURE (VAR sc: Scanner) **Scan**
Scans the ASCII text file for the next token. It skips over 'White space'; typically spaces (20X), tabs (09X), carriage returns (0DX), linefeeds (0AX), and control characters (characters before space in the ASCII character set).

When it finds non-white characters it (generally) collects the largest contiguous section together, and this stretch of non-white characters is called a 'token'. The nature (type) of the token is then ascertained (eg one of INTEGER, LONGINT, REAL, BOOLEAN, SET, Symbol (a CHAR), or String), and the token, its type, and its 'value' are recorded in the Scanner.

If there are no more tokens on the current line *sc.type* is set to *isEol* unless *freeFormat* is in *sc.opts*.

If *freeFormat* is in *sc.opts* further lines are read from the file as required. If there are no more tokens in the file *sc.type* is set to *isEof*.

Tokens may contain spaces only if
→ they are enclosed in quotation marks (matching pairs of ' or ") and *useQuotes* is in *sc.opts*.
or
→ *sc.opts* does not contain *spcSep*, which is only permitted if it does contain *commaSep* or *tabSep*.

PROCEDURE (VAR sc: Scanner) **Set** (): SET
Scans for the next token, and tries to interpret it as a SET value.
(Note that spaces are not permitted in tokens if *sc.opts* contains *spcSep*, which it does by default, unless the token is enclosed in quotes and *useQuotes* is in *sc.opts*. .)

If the token is not acceptable (which requires *sc.opts* to contain *seekSet* and not *seekSym*) *sc.type* is set to *invalid*.
In addition, if *sc.opts* contains *typeTrap*, the programme traps (TRAP 85).

If *sc.type* # *isSet sc.typeErr* is set TRUE.
Note that this includes the cases *sc.type* = *isEol* & *sc.type* = *isEof*, which do not cause a Trap.

PROCEDURE (VAR sc: Scanner) **SetColumn** (col: INTEGER)
Sets the position of the Scanner in the current line.

Note: the full state of the Scanner is not reset. *sc.type* may be set to *invalid*. It is updated after the next Scan.

Pre
25 0 <= col <= LEN (sc.line)

PROCEDURE (VAR sc: Scanner) **SetOpts** (opts: SET)
Sets the Scanner's options.

These are:
tabIsSpace, *trimSpc*, *spcSep*, *commaSep*, *tabSep*, *useQuotes*, *seekBool*, *seekSym, stopIsSym, seekSet*, *lcExponent*, *freeFormat*, & *typeTrap*.

If *tabIsSpace* is set the option *tabSep* is irrelevant, because there will be no Tabs in *sc.line*.

If *spcSep* is set the option *trimSpc* is irrelevant, because there will be no spaces in *sc.token* to trim.

If neither *commaSep* or *tabSep* is set the option *spcSep* is automatically set even if it is omitted from *opts*.

If *seekSym* is set the option *seekSet* is irrelevant, because no SETs can be parsed.

PROCEDURE (VAR sc: Scanner) **SetPos** (pos: INTEGER)
Sets the scanner 'position' to *pos*. For more explanation refer to Pos above.

This routine should be used with caution because:

► If the scanner is connected to a File the file is re-read from the beginning, which may be slow.

► If the scanner is connected to a Text this routine jumps to near the correct position, then does a local search. This may be significantly faster than the technique above. However it does NOT necessarily set the values *sc.lineNo*- & *sc.tknNo* correctly. Hence it should not be used if these values are required.

PROCEDURE (VAR sc: Scanner) **String** (): Str
Scans for the next token, and returns it as a string value.
If a token is found *sc.type* is set to *isStr*.

If *sc.type* = *isEol* or *sc.type* = *isEof* *sc.typeErr* is set TRUE.

PROCEDURE (VAR sc: Scanner) **Symbol** (): CHAR
Scans for the next token, and tries to interpret it as a Symbol value, ie one of the CHARs listed under *seekSym*.

If the token is not acceptable (which requires *sc.opts* to contain *seekSym*) *sc.type* is set to *invalid*.
In addition, if *sc.opts* contains *typeTrap*, the programme traps (TRAP 85).

If *sc.type* # *isSym* *sc.typeErr* is set TRUE.
Note that this includes the cases *sc.type* = *isEol* & *sc.type* = *isEof*, which do not cause a Trap.

```
Author   :  Robert D Campbell
Updated  :  10  August  2012
```