The `Scan` object — TCC

The `Scan` object was invented to make it easier to write small compilers and other applications that syntax-check files. Its purpose is to take care of a lot of the mechanical details of scanning, and especially to make it easy to produce error messages that identify the file, line, and exact location of the error.

However, the Icon language already has a large and well-designed set of scanning primitives, and we wouldn't want to have to reinvent all those mechanisms. So the `Scan` object works along with native Icon scanning. We assume that the reader is familiar with the Icon scanning operator (`?`), the built-ins `&subject` and `&pos`, and the various scanning functions such as `any()`, `tab()`, `move()`, and so forth. See *The Icon Programming Language* by Griswold for complete details, or see this URL for a tutorial:

`http://www.nmt.edu/tcc/help/lang/icon/`

The `Scan` object also supports an optional line-comment character. For example, you might ask for the `Scan` object to ignore a percent sign (`%`) and everything after it on that line.

Often I find that I want my error messages to identify the site of an error with more information than just the file name and line number. The `Scan` object also allows you to provide a callback procedure that can furnish a string that identifies where you are in the file. For example, suppose you are writing a compiler for a language that divides code into modules. The callback procedure might identify which module you are currently in.

Basic methods

To scan a file using the `Scan` object, first use `Scan_Open()` to point it at some readable text file. This method will make the first line of the file the current scan subject (`&subject`) and set the position (`&pos`) to 1.

Then you can use the usual Icon scanning primitives to move and scan items from the subject. To move to the next line, you can use the `Scan_Next_Line()` method or several other methods that can bring up a new line as the scan subject.

The `Scan` object makes heavy use of the `Log` object to manage the error messages that you may issue. This object is described elsewhere; see the URL

`http://www.nmt.edu/~john/soft/clean/lib.html`

for documentation of all my standard objects.

`Scan_Open`

To open a `Scan` object and associate it with a file:

```
Scan_Open ( fileName, log, commentChar, callback )
```

where the arguments have these definitions:

`fileName`	The name of the file to be scanned.
`log`	A `Log` object that will receive any error messages from scanning this file.

`commentChar`	If given, this argument should be a string containing the comment character. The default is not to have a comment character.
`callback`	If given, this is a procedure with calling sequence `callback(scan)`, where `scan` is a `Scan` object, that returns a string that is appended to an error message each time one is issued.

The `Scan_Open()` method returns a `Scan` object that must be passed as the first argument to the remaining methods.

`Scan_Close`

This method should be called when scanning is completed:

```
Scan_Close ( scan )
```

where `scan` is the object returned by `Scan_Open()`.

`Scan_End_File`

To test whether there is any text left in a `Scan` object:

```
Scan_End_File ( scan )
```

This method succeeds and returns `&null` if `scan` is at end of file, or fails if there is still some text left in the file.

`Scan_End_Line`

This method succeeds and returns `&null` if `scan` is at the end of the current line, or fails if there is text left on the line. Calling sequence is:

```
Scan_End_Line ( scan )
```

`Scan_Error`

To issue an ordinary error message, use this call:

```
Scan_Error ( scan, text, ... )
```

where the `text` arguments are strings that are concatenated to form the final error message.

The output text will look like this:

`*** Error:` *message*

Here's an example:

```
Scan_Error ( scan, "I have now taunted you ", nTaunts, " times." );
```

The `Scan_Error()` method always fails, which allows you to telescope the usual construct

```
if something bad happens then
{ Scan_Error ( scan, "Bad human. No donut." );
  fail;
}
```

into the shorter version

```
if something bad happens then
  return Scan_Error ( scan, "Bad human. No donut." );
```

which relies on the property of Icon that if an expression in a `return` statement fails, that is the same as evaluating the expression and then `failing`.

`Scan_Error_Kind`

Normally all errors are prefaced by the string `"Error:"`. However, you can define multiple kinds of errors—for example, errors, warnings, and informational messages—and keep separate counts of the errors in each category. To issue a message in a non-error category, use:

```
Scan_Error_Kind ( self, kind, text, ... )
```

where `kind` is a string that describes the error category. To issue a warning, for example, you might use a call like this:

```
Scan_Error_Kind ( self, "Warning", "Your mileage may vary." );
```

`Scan_Error_Count`

You can find out how many errors have been issued to a `Scan` object by using:

```
Scan_Error_Count ( scan, kind )
```

The error count is returned as an integer; if there have been no errors sent to the object, you will get a zero back.

If you omit the `kind` argument you will get a count of the messages of the default kind, which is `"Error"`.

Application. The obvious application for `Scan_Error_Count()` is printing messages such as "You had 9 errors and 3 warnings." However, sometimes I find it convenient for diffusing error reporting while centralizing error detection.

For example, suppose you have a large section of code that checks for a bunch of different errors, and any one of several dozen procedures might generate an error. One way to handle it would be for each procedure to fail if it detected an error. There are two disadvantages to this:

- Every call to a procedure must check for failure and take appropriate action.
- If you fail on the first error, you may miss a chance to detect other errors.

So another way to handle this is to record the error count before executing that section of code, and then check the error count afterwards to see if it has gone up. If so, you know there has been an error somewhere. Here is how that code might look:

```
procedure Sample ( scan )
  local errCount

  errCount := Scan_Error_Count ( scan );

  Big_Procedure ( scan );     # Errors can come anywhere here

  if  Scan_Error_Count ( scan ) > errCount  then
    fail
  else
    return;
end
```

`Scan_Next_Line`

If you are doing line-oriented processing, this method will advance the `Scan` object to the beginning of the next line in the input file, if there is one:

```
Scan_Next_Line ( scan )
```

If no lines remain in `scan`, the method fails, otherwise it returns `&null`.

Handy scanning primitives

The methods below are provided to handle common operations.

`Scan_Deblank`

If your file has a free-field format that treats line breaks as whitespace, you can skip all whitespace on the current line and any following lines to get to the next nonblank character by calling:

```
Scan_Deblank ( scan, whiteCset )
```

where `whiteCset` is an optional Icon `cset` defining which characters are considered whitespace. Newline is always considered whitespace. Use `Scan_End_File()` to find out if there is any text left after `Scan_Deblank()`. The default value of `whiteCset` is `' \t'`, that is, spaces and tabs.

`Scan_Equal_Arb`

This method is like the Icon built-in unary `=` operator, except that it is case-insensitive.

```
Scan_Equal_Arb ( scan, s )
```

The `s` argument must be a nonempty string. If the subject starts with a string that is identical to `s` using case-insensitive comparison, `&pos` is advanced past that matching string and the matching string is returned. If the subject does not start with a matching string, `Scan_Equal_Arb()` fails.

`Scan_Fixed`

To scan a fixed-point constant (such as 3.1415 or 0.007), call this method:

```
Scan_Fixed ( scan )
```

If the current line in `scan` starts with a digit, `Scan_Fixed()` advances the position past the constant and returns its value as a real. The method fails if there is no digit at `&pos`.

This method requires that there be at least one digit before the decimal point. This is because when I see something that looks like `.1` on a printout, I tend to wonder whether that's really 0.1 or just a 1 with a flyspeck in front of it. Integers are considered legal fixed-point constants; no decimal is required. For example, if the `Scan` object starts with the string `457`, `Scan_Fixed()` will consider it valid and return 457.0 as a real.

`Scan_Flat_Cset`

If you are processing a flat file, where each field has a fixed size, it is often convenient to check that the next n characters of the line are all in a particular set of characters:

Scan_Flat_Cset (scan, n, c)

where n is the width of the field in characters and c is an Icon `cset` defining the valid characters. If the next n characters of `scan` are all in c, `scan` is advanced by n characters and the result returned is the contents of the field. If there aren't at least n characters left, or the next n characters contain some that are not in c, the method fails without advancing `scan`.

Scan_Flat_Int

Another common need when scanning flat files is to process an integer that is right-justified in a fixed field:

```
Scan_Flat_Int ( scan, n )
```

where n is the width of the field in characters. The syntax of a number is given by

```
<int> ::= [ " " ...  ]  [ "-" ] [ " " ...  ]  digit ...
```

That is, the digits must be left-padded with blanks, and an optional minus sign may appear anywhere in that blank-padded area. If the next n characters of `scan` are valid according to this syntax, `scan` is advanced by n characters and the method returns the signed number as type `integer`. If there do not remain at least n characters on the current line, or the next n characters don't fit the syntax, the method fails without advancing `scan`.

Scan_Gen_Lines

If you would like to iterate over the lines of a file, use this construct:

```
every Scan_Gen_Lines ( scan ) do ...
```

This construct executes the body of the loop once for each line in the `Scan` object, making each line the subject in turn.

This method also returns the lines in case you want them:

```
every line := Scan_Gen_Lines ( scan ) do ...
```

Scan_Integer

To parse an integer (that is, a string of digits), call:

```
Scan_Integer ( scan )
```

If `scan` starts with at least one digit, `&pos` will be advanced past all leading digits and the number will be returned to you as an integer. If the next character in the current line is not a digit, the method fails.

Scan_Line_Number

To find out the line number of the current line in a `Scan` object:

```
Scan_Line_Number ( scan )
```

This method returns the number as an integer, counting from 1.

Scan_Match_Arb

This method is similar to the Icon built-in function `match()`, but it uses case-insensitive comparison:

```
Scan_Match_Arb ( scan, s )
```

The `s` argument is a string. If the next `(*s)` characters of the subject match `s`, disregarding case, this method returns the position of the end of the matching part. If there is no match, the method fails.

Intended functions

More information about my particular variant of the Cleanroom software development methodology can be found at:

http://www.nmt.edu/~john/soft/clean

Here are the Cleanroom intended functions for the above methods.

```
#--
# Scan_Open (       # Open a stream for scanning
#   fileName,       # Stream file name, or & null for & input
#   log,            # Error logging object; see log.icn
#   commentChar,    # Comment char, or & null for none
#   callback )      # Localization callback procedure (see Scan_Error)
#       [ if fileName is readable ->
#           returns a scan object pointing at the start of that file,
#           logging to (log), using (commentChar) as the line comment
#           character if given, and using (locProc) as the callback
#           if given
#         else ->
#           fail
#       ]
#--
# Scan_Close ( self ): Close stream self
#       [ closes self's file, if it isn't already closed
#         returns & null
#       ]
#--
# Scan_End_File ( self ):
#       [ if self is at end of file ->
#           return & null
#         else -> fail
#       ]
#--
# Scan_End_Line ( self ):
#       [ if self is at the end of the current line ->
#           return & null
#         else -> fail
#       ]
#--
# Scan_Error ( self, text, ... ):
#       [ if the (text) arguments are all strings ->
#           same intended function as Scan_Error_Kind but with a
#           default value for 'kind'
#       ]
#--
```

```
#--
# Scan_Error_Count ( self, kind )
#       [ if (kind) is a string or & null ->
#           returns the count of errors of kind (kind) from self's log
#           object, where (kind) may be & null for the default
#           error kind
#       ]
#--
# Scan_Error_Kind ( self, kind, text, ... );
#       [ if (kind) is a string or & null, and the (text) arguments are
#         all strings ->
#           if this is the first error for the current line ->
#             self's log ||:= a message showing the file name,
#                             line number, and the result of self's
#                             locProc(self) if any, followed by
#                             an echo of self's current line
#         In any case:
#           self's log  ||:=  message showing a carat at the position
#                             corresponding to & pos, followed by an
#                             error of class 'kind' whose message text
#                             is the concatenation of all 'text' arguments
#       ]
#--
# Scan_Gen_Lines ( self )
#   [ generates the lines from self  ]
#--
# Scan_Line_Number ( self )
#   [ returns the line number of the current line in self, counting from 1
#   ]
#--
# Scan_Message ( self, text, ...)
#   [ if the (text) arguments are all strings ->
#       self's log  ||:=  message containing the concatenation
#                         of all text args
#   ]
#--
# Scan_Next_Line ( self ):
#       [ if self is at end of file -> fail
#         else if no lines remain in self ->
#           close self's file
#           fail
#         else ->
#           & subject  :=  the next line from self's file
#           return & null
#       ]
#--
```

```
#--
# Scan_Deblank ( self, wset )
#       [ let whitespace = an element of wset, if given, or ' '
#                          (that is, space and tab) if it is & null
#         in:
#         if wset is a cset or & null ->
#           if self is not at EOF and contains at least one non-whitespace
#           character ->
#             self  :=  self advanced past any leading whitespace
#                       characters, skipping lines if necessary
#                       until a non-whitespace character is found
#             return self
#           else ->
#             fail
#       ]
# Note: to deblank without skipping to the next line, use this construct:
#   tab(many(wset))
#--
# Scan_Equal_Arb ( self, s )
#       [ if self starts with (s), ignoring case ->
#           self  :=  self advanced by the length of (s)
#           return the next (length of s) characters
#       ]
# Note: This function is like Icon's unary "=" operator, only it's
# case-insensitive.
#--
# Scan_Find_Arb ( self, s )
#   [ if s is a nonempty string ->
#       generate the positions in the current line of self where s
#       occurs (case-insensitive)
#   ]
# Note: This is like the Icon function find(), only case-insensitive.
#--
# Scan_Fixed ( self )
#   [ let <fixed>  have the syntax
#       <digit>  ... [ "." [ <digit>  ] ... ]
#     in:
#     if self starts with a <digit>  ->
#       self  :=  self advanced past the leading <fixed>
#       return the <fixed> , represented as a real.
#     else ->  fail
#   ]
# Note: This routine is for scanning fixed-point numbers.  Note that it
# does not allow a number to start with a decimal.  I find it too hard
# to distinguish ".1" from a "1" preceded by a flyspeck, so I require
# that it be typed is "0.1".
#--
```

```
#--
# Scan_Integer ( self, maxLen )
#   [ if maxLen is an integer or & null ->
#       if (maxLen) is & null, or the line in self starts with a digit
#       but not more than (maxLen) of them ->
#         self  :=  self advanced past all leading digits
#         return those digits, as type integer
#       else if (maxLen) is given and the line in self starts with more
#       than (maxLen) digits ->
#         self  :=    self advanced past all leading digits
#         self  ||:=  error, integer too long, maximum is (maxLen)
#         return the first (maxLen) digits from self
#   ]
#--
# Scan_Match_Arb ( self, s )
#   [ if s is a nonempty string ->
#       if string (s) is a prefix of the current line in self, using
#       case-insensitive comparison ->
#         return the position in that line where that string ends
#       else ->  fail
#   ]
# Note: Like the Icon match() function, only case-insensitive.
#--
# Scan_Flat_Cset ( self, n, c )
#   [ if n is a positive integer and c is a cset ->
#       if self begins with n characters in c ->
#         self  :=  self advanced by n characters
#         return the next n characters of self
#       else ->  fail
#   ]
#--
# Scan_Flat_Int ( self, n )
#   [ if n is a positive integer ->
#       if self starts with an integer of size n, left-padded
#       with blanks, and with a possible "-" sign located
#       anywhere within those blanks ->
#         self  :=  self advanced by n characters
#         return that signed integer as type integer
#       else ->
#         fail
#   ]
#--
```

Written by John W. Shipman (john@nmt.edu). This version printed 1997-09-24.