Lexical conventions:

The input character stream is split into tokens as it is processed. Each token consists of a sequence of non-whitespace characters and (possibly) trailing whitespace characters. Whitespace is defined to be space, tab, carriage return, or newline - non-space characters are to be interpreted at formatting time, but internally are treated as identical. Each token ends at the non-whitespace character following previous whitespace, or at one of the brackets (by default, < and >). The brackets themselves are treated as separate tokens, but are removed during the evaluation process (see below). If the quote character appears (by default, \), the next character (including whitespace) will lose any special meaning it may have, and will become part of the token; however, once such a token is evaluated, the quote is removed. Generally, this is of no consequence (see evaluation). Another special character is defined, the reference character (by default, ~), which brackets references in text, and is actually macro-expanded by the scanner to `.link < ... > < ... >` (e.g., `~Reginald Maudling~` -> `.link <Reginald Maudling> <Reginald Maudling>`). Finally, the argument character (by default, @) is used to represent arguments to macros (see below). Under certain circumstances, resetting bracket characters (and possibly the quote character) could cause a great deal of confusion, so these should probably be restricted to the beginning of the file.

The bracket characters serve several purposes. One is to collect multiple tokens, to be treated as one entity (e.g., "`<a b>`"); another is to separate tokens without intervening whitespace (e.g., "`Thus ended .user's-name<'s> life.`"); the last is to delimit the argument list of a command (see next). It should be noted that bracketing strings will NOT prevent splitting over a line - for this, the \ character should precede each blank, e.g., "`Elmer\ Fudd`" would be guaranteed to be displayed on one line.

Commands are tokens beginning with the command character (by default, .) AND which contain other non-blank characters (thus, a lone . will not be treated as a command). Any undefined variable names are passed verbatim, so, in general, numbers beginning with decimal points, and ellipses are treated correctly, without special attention. However, it will be possible to detect undefined names so that an author may be warned of their presence. A command is followed by its arguments, which are individual tokens or bracketed collections of tokens. The argument list of a command terminates at the next command OR closing bracket at the same nesting level.

An example:

".foo a b .bar c d" <->

≡ "<.foo a b> <.bar c d>" <->

≡ ".foo <a> <b> .bar <c> <d>" <->

≡ "<.foo <a> <b>> <.bar <c> <d>>", but is NOT equivalent to, for example:

".foo <a b> .bar c d" or, even worse,

".foo <a b <.bar c d>>", which says that the sole argument to .foo is <a b <.bar c d>>.

Capabilities:

Commands, as described above, are really a more general mechanism than it may appear. The name of a command, first of all, is the command token, converted to lowercase, with leading, trailing, and duplicate internal blanks removed. In evaluating a command, the name is looked up, and is either replaced by a string of tokens, or the result of executing a built-in function.
A mechanism exists for associating arbitrary names with token strings, so that a "command" can actually be a variable, and furthermore, such variable token strings may define macros with arguments. This means that the functionality of a built-in command may be replaced or modified by creating an appropriate macro string. Also, command/macro/variable assignments may be performed locally or globally (see below). Finally, the result of a collection of tokens or commands may be invoked as a command with the form ".< ... >".

Macros with arguments require the use of a special built-in that looks ahead in the token stream and "cuts" a specified token, returning that as its value. To illustrate:

```
.set swap <@2 @1>
...
.swap hey you
        ->      <@2 @1> hey you
        ->      you hey

or,
.set blink <.link @2 @1>
...
.blink <George Washington> <George, baby!>
        ->      .link @2 @1 <George Washington> <George, baby!>
        ->      .link <George, baby!> <George Washington>>
        ->      ... whatever
```

Another important facility is the inclusion of files within files. The command

```
.include file
```

causes the file named by "file" to be read in at that point in the input, exactly as if it were part of the original file being processed. It is important to note that the file may be named by a variable, e.g.:

```
.set file <the mighty herring>
...
.include <.file>
        -> .include <<the mighty herring>>
```

(in this language, `<<string>>` `<->` `<string>` `<->` `string`; only argument grouping is affected).

Practical Usage:

The formatter will need to be used both as a driver for the display of formatted text, and as a tool for extracting particular pieces of information from a file (during index or xref generation). Also, there is considerable utility to being able to specify default formatting options, etc. The include processing capability can accomplish both of these. The low-level commands for setting formatting parameters and writing text to the display driver can be designed to be accessible as variables, as can the functions for extracting storyboard attributes, error message printers, etc. This means that an outer wrapper can reset these variables such that various outputs can be disabled or diverted, and the formatter can be used for numerous purposes. This in turn saves on rewriting the same parser several times, or having to keep several software revisions in step.

The macro utility can also be used to simplify the language for naive users, by establishing default command precedences. This is done by including extra bracket characters in the output of a macro. To illustrate:

The commands .a and .b are to be defined so that .a has higher precedence (binds more tightly than) .b. Now, assume "a" has the value "> .A <", and "b" has the value ">> .B <<". Then,

```
.a ... .b ...
        ->      > .A < ... >> .B << ...
```

whereas

```
.b ... .a ...
        ->      >> .B << ... > .A <
```

which results in the desired precedence.

This glosses over some problems with making sure that .A gets the right arguments, but these can be overcome with @ substitution. Also, provision must be made for balancing the extra brackets that will occur at the ends -- one possibility is an ".enclose" command which sloughs off any excess brackets at the beginning and end of an input file.

Evaluation:

The top-level of formatter execution is a reader function (basically, ".enclose" as above). This continually reinvokes a printing function, Print (using current formatting parameters), until the end of file is reached. Now, the execution of the formatter is best described by tracing the execution of formatter functions, beginning with Eval.

Eval: Get (and remove) the next token from the input stream. Depending on the type of this token, various actions may be taken. The cases are as follows:

- ordinary text string - remove embedded quotes, and return the string to caller
- left bracket - recursively call Eval
- command - call Command, to process the command, and leave the input stream ready to continue.
- argument - call Argument, to find the argument(s), and leave its/their value in the input stream.
- right bracket - return an empty string

Command: Call Eval to get the value of the next expression (which will usually be a simple text string, but could be a complex expression that evaluates to a simple text string). Look up this value in the symbol table, and if defined as a string, push back its value onto the input, if defined as a built-in function, call the appropriate function, else if undefined, push back the string preceded by a quoted command character. Return.

Argument: Again, call Eval. The returned value should be a string, which should be a decimal representation of an integer, or an integer followed by a dash. These represent the positions of the arguments in the input stream, AT ONE NESTING LEVEL HIGHER than that where Argument was called. Arguments are numbered starting with 1. The dash means from the specified argument to the last (non-empty) argument in the expression (treated as one string). Arguments past the end of the argument list (delimited by the right bracket or another command) are empty strings. All arguments through the last one referenced are cut from the input. Argument uses a low-level function for scanning the token stream and extracting the nth expression (at a given nesting level). These are bound to variables internal to the caller (either Eval, or a built in function), so they may be referenced more than once. None of these arguments are evaluated here. The result is pushed back onto the input.

Built-in Functions: These generally work by setting some internal parameters, sometimes based on arguments supplied. A function may use its arguments unevaluated, or evaluate any of them by pushing them back (with brackets) and calling Eval. Generally, there follow arguments that are to be printed, rather than consumed. This is accomplished by calling Print, which again invokes Eval repetitively until the argument list is exhausted. Some cleanup, or resetting of internal parameters may be performed, and then finally, a result may be pushed back onto the input.