

1. Introduction.

ETSET
Typeset an Electronic Text

by John Walker

This program is in the public domain.

```
#define PRODUCT "etset"  
#define VERSION "3.0.1"  
#define REVDATE "2001-09-28"
```

2. Command line.

ETSET is invoked with a command line as follows:

```
etset options input_file output_file
```

where *options* specify processing modes as defined below and are either long names beginning with two hyphens or single letter abbreviations introduced by a single hyphen. If no *input_file* is specified, or “-” is given for its name, input is read from standard input. Similarly, output is written to standard output if *output_file* is omitted or “-” is specified. When generating HTML, an *output_file* name *must* be specified; it is the “base name” used to generate the various HTML files making up the document tree, which are created in the current directory.

3. Options.

Options are specified on the command line prior to the input and output file names (if any). Options may appear in any order. Long options beginning with “--” may be abbreviated to any unambiguous prefix; single-letter options introduced by a single “-” may be aggregated.

- ascii-only** Check for the presence of any characters not part of the 7-bit ASCII set (for example, accented letters belonging to the ISO 8859-1 set), and generate warning messages identifying them.
- babel *lang*** Use the L^AT_EX **babel** package for language *lang*.
- check** Check text for publication. Report any invalid characters or formatting errors to standard error.
- clean** Clean up text for publication: expand tab characters to spaces, remove trailing blanks from lines.
- copyright** Print copying information.
- debug-parser *file*** Write parser debugging information to *file*. Each line in the body of the text is labeled with the identification assigned it by the parser.
- dos-characters** Translate MS-DOS Code Page 850 character set to ISO 8859-1 and remove carriage returns from the ends of lines.
- flatten-iso** ISO 8859-1 8-bit characters are replaced with their closest 7-bit ASCII equivalent (for example, accented letters are changed to unaccented characters). This is a *destructive* transformation, and should be performed only when a text must be displayed on a device which cannot accept 8-bit characters.
- french-punctuation** Insert nonbreaking spaces around punctuation as normally done when typesetting French. Guillemets, colons, semicolons, question marks, and exclamation points are set off from the adjoining text by a space. This mode is unnecessary when typesetting French with the “**--babel francais**” option.
- help, -u** Print how-to-call information including a list of options.
- html, -h** Generate HTML output. By default, a document tree is generated with an index document which links to individual chapter documents, each of which contains navigation links. If the **--single-file** option is specified, a single HTML document containing the entire text is generated. HTML files are written to the current directory.
- latex, -l** Generate a L^AT_EX file to typeset the document. If the document is in a language other than English, you may also wish to use the **--babel** option to invoke formatting appropriate for the language.
- palm, -p** Generate a file in Palm Markup Language to create a document for Palm Reader on handheld platforms.
- save-epilogue *file*** The document epilogue is written to the designated *file*.
- save-prologue *file*** The document prologue is written to the designated *file*.

- single-file** Generate a single HTML file containing all chapters, as opposed to the default of a document tree with a separate file for each chapter.
- special-strip** Remove all format-specific special commands from the document, and blank lines following special command if they would result in consecutive blank lines in the document. This option may be used in conjunction with the **--clean** option when preparing a text for publication in “Plain ASCII” format.
- verbose, -v** Print information regarding processing of the document, including the number of lines read and written.
- version** Print program version information.

4. Input format.

Beautifully Typeset Etexts

Plain Vanilla Etexts don't have to be austere and typographically uninviting. Most literature (as opposed to scientific publications, for example), is typographically simple and can be rendered beautifully into type without encoding it into proprietary word processor file formats or impenetrable markup languages. Etexts may be encoded in a form which permits them to be both read directly (Plain Vanilla) and typeset in a form virtually indistinguishable from printed editions of the work.

To create “typographically friendly” Etexts, observe the following rules:

1. Characters follow the 8-bit ISO 8859/1 Latin-1 character set. ASCII is a proper subset of this character set, so any “Plain ASCII” file meets ths criterion by definition. The extension to ISO 8859/1 is required so that Etexts which include the accented characters used by Western European languages may continue to be “readable by both humans and computers”.
2. No white space characters other than blanks and line separators are used (in particular, tabs are expanded to spaces).
3. The text bracket sequence:

`<><><><><><><><><><><><><><><><><><>`

 appears both before and after the actual body of the Etext. This allows including an arbitrary prologue and epilogue to the body of the document.
4. Normal body text begins in column 1 and is set ragged right with a line length of 70 characters. The choice of 70 characters is arbitrary and was made to avoid overly long and therefore less readable lines in the Plain Vanilla text.
5. Paragraphs are separated by blank lines.
6. Centring, right, and left justification is indicated by actually so-justifying the text within the 70 character line. Left justified lines should start in column 2 to avoid confusion with paragraph body text.
7. Block quotations are indented to start in column 5 and set ragged right with a line length of 60 characters.
8. Preformatted tables begin with a line which starts in column 3 and contains at least one sequence of three or more spaces between nonblanks. The table is formatted verbatim until the next blank line.
9. Text set in italics is bracketed by underscore characters, “_”. These must match.
10. Footnotes are included in-line, bracketed by “[]”. The footnote appears at the point in the copy where the footnote mark appears in the source text. Footnotes may not be nested and may consist of only a single paragraph.
11. The title is defined as the sequence of lines which appear between the first text bracket “<><><...” and a centred line consisting exclusively of three or more equal signs “====” .
12. The author’s name is the text which follows the line of equal signs marking the end of the title and precedes the first chapter mark. This may be multiple lines.
13. Chapters are delimited by a three line sequence of centred lines:

[illegible]

appears both before and after the actual body of the Etext. This allows including an arbitrary prologue and epilogue to the body of the document.

Normal body text begins in column 1 and is set ragged right with a line length of 70 characters. The choice of 70 characters is arbitrary and was made to avoid overly long and therefore less readable lines in the Plain Vanilla text.

5. Paragraphs are separated by blank lines.

6. Centring, right, and left justification is indicated by actually so-justifying the text within the 70 character line. Left justified lines should start in column 2 to avoid confusion with paragraph body text.

7. Block quotations are indented to start in column 5 and set ragged right with a line length of 60 characters.

8. Preformatted tables begin with a line which starts in column 3 and contains at least one sequence of three or more spaces between nonblanks. The table is formatted verbatim until the next blank line.

9. Text set in italics is bracketed by underscore characters, “_”. These must match.

10. Footnotes are included in-line, bracketed by “[]”. The footnote appears at the point in the copy where the footnote mark appears in the source text. Footnotes may not be nested and may consist of only a single paragraph.

11. The title is defined as the sequence of lines which appear between the first text bracket "<><>..." and a centred line consisting exclusively of three or more equal signs "====".

12. The author's name is the text which follows the line of equal signs marking the end of the title and precedes the first chapter mark. This may be multiple lines.

13. Chapters are delimited by a three line sequence of centred lines:

Chapter number

Chapter name

The line of minus signs must be centred and contain three or more minus signs and no other characters apart from white space. Chapter “numbers” need not be numeric—they can be any text.

14. Dashes in the text are indicated in the normal typewritten text convention of “--”. No hyphenation of words at the end of lines is done.

18. Quote marks may be rendered explicitly as open and close quote marks with the sequences ‘single quotes’ or “double quotes”. As long as quotes are balanced within a paragraph, the ASCII quote character ‘”’ may be used. Alternating occurrences of this character will be typeset as open and close quote characters. The open/close quote state is reset at the start of each paragraph, limiting the scope of errors to a single paragraph and permitting “continuation quotes” when multiple paragraphs are quoted.

5. Program global context.

⟨Preprocessor definitions⟩
 ⟨System include files 145⟩
 ⟨Program implementation 6⟩

6. The following classes are defined and their implementations provided.

⟨Program implementation 6⟩ ≡
 ⟨Global variables 48⟩
 ⟨Global functions 147⟩
 ⟨Class definitions 8⟩
 ⟨Main program 141⟩

This code is used in section 5.

7. The following definitions describe the formatting of input body copy. Note that column numbers cited below assume the first column of a line is 0.

```

#define FormatWidth 70 /* Format width of original text */
#define RaggedRightIndent 1 /* Indentation for ragged right copy */
#define PreformattedTableIndent 2 /* Indentation for preformatted tables */
#define QuoteIndent 4 /* Indentation for block quotes */
#define TitleMarkerCharacter '=' /* Character identifying document title/author sequences */
#define ChapterMarkerCharacter '-' /* Character identifying chapter number/title sequences */
#define MarkerMinimumLength 3 /* Minimum length of title and chapter markers */
#define SpecialMarker "<><><>" /* Special text line marker (start and end of line) */
#define SpecialPrefix (SpecialMarker "Special:") /* Special text line prefix */
#define PUNCTUATION ("?!:;" RIGHT_POINTING_DOUBLE_ANGLE_QUOTATION_MARK)
/* Punctuation set after a space in French text processed with the -f option */
#define Iabs(x) (((x) < 0) ? (-(x)) : (x)) /* Absolute value */

```

8. Text processing components.

The *textComponent* class is the abstract superclass of all of the text source, sink, and filter classes. A source is simply a filter whose input is not a component, and a sink a filter whose output is not a component.

⟨ Class definitions 8 ⟩ ≡ /* This ought to be a **static** member of *textComponent*, but I'll be damned if

I can figure out how to make it work as one. */

```
static const string fTypeName[4] = {"Undefined", "Source", "Filter", "Sink"};
```

```
class textComponent {
```

```
protected:
```

```
    textComponent *output;    /* Next filter in chain */
```

```
    textComponent *source;    /* Source at head of pipeline */
```

```
    int lineNumber;    /* Output line number */
```

```
    enum filterType {
```

```
        UndefinedType = 0, SourceType = 1, FilterType = 2, SinkType = 3
```

```
    };
```

```
    filterType fType;
```

```
public:
```

```
    textComponent()
```

```
    {
```

```
        output = Λ;
```

```
        source = Λ;
```

```
        lineNumber = 0;
```

```
        fType = UndefinedType;
```

```
    }
```

```
    virtual string componentName(void) = 0;    /* Return name of filter */
```

```
    virtual void put(string s) = 0;    /* Write string to filter */
```

```
    ⟨ Connect components in pipeline 9 ⟩;
```

```
    ⟨ Emit output to next component in pipeline 10 ⟩;
```

```
    ⟨ Handle end of file notification 11 ⟩;
```

```
    textComponent *getSource(void)
```

```
    {
```

```
        assert(source ≠ Λ);    /* Filter not wired to a source */
```

```
        return source;
```

```
    }
```

```
    int getLineNumber(void)
```

```
    {    /* Output line number of this filter */
```

```
        return lineNumber;
```

```
    }
```

```
    int getSourceLineNumber(void)
```

```
    {    /* Line number of ultimate source */
```

```
        return getSource()-getLineNumber();
```

```
    }    /* Issue message tagged with source line number */
```

```
    virtual void issueMessage(string msg, ostream &of = cerr)
```

```
    {
```

```
        of << getSourceLineNumber() << ":␣" << msg << "\n";
```

```
    }    /* Write description to stream of */
```

```
    virtual void writeDescription(ostream &of)
```

```
    {
```

```
        of << fTypeName[fType] << ":␣" << componentName() << "\n";
```

```
    }
```

```
;
```

```
};
```

See also sections [12](#), [13](#), [14](#), [15](#), [16](#), [17](#), [18](#), [19](#), [21](#), [23](#), [24](#), [27](#), [28](#), [42](#), [44](#), [45](#), [46](#), [47](#), [51](#), [52](#), [62](#), [63](#), [64](#), [66](#), [67](#), [68](#), [69](#), [76](#), [77](#), [90](#), [91](#), [92](#), [102](#), [103](#), [105](#), [112](#), [113](#), [114](#), [115](#), [116](#), [117](#), [119](#), [120](#), [127](#), [128](#), [129](#), and [140](#).

This code is used in section [6](#).

9. Every text processing pipeline must start with a source (*fType* of *SourceType*) and terminate with a sink (*fType* of *SinkType*). Any number of filters (*fType* of *FilterType*) may be interposed between these ends. Successive components in a pipeline are connected to one another by calling the *setOutput* method of each component, starting with the source, in the order in which they appear in the pipeline, giving the next component as the argument. The sink at the end of the pipeline delivers the result to the ultimate output, so *setOutput* is not called for it.

For example, suppose you have a three component pipeline consisting of a source named **faucet**, a filter **strainer**, and a sink **sewer**. To plumb these three components into a pipeline, you could make the following function calls:

```
faucet.setOutput(strainer);
strainer.setOutput(sewer);
```

We overload the `|` operator to allow connecting pipelines in a less verbose fashion, one familiar to users of UNIX shell commands. Using this operator, the three component pipeline can be connected with the single expression:

```
faucet | strainer | sewer;
```

Each component in a pipeline contains a link back to the first (*SourceType*) component. The source component points to itself, and the link to the start of the pipeline is propagated as each additional component is added. To obtain the source link, use the *getSource* method of any component in the pipeline. This is frequently used when a downstream component wishes to label a diagnostic message with the line number of the original source line from which the text it's processing was derived. This is needed so frequently, in fact, that the *getSourceLineNumber* method is provided to directly obtain this value.

⟨Connect components in pipeline [9](#)⟩ ≡

```
virtual void setOutput(textComponent &ofilt)
{
    output = &ofilt;
    ofilt.source = source;    /* Propagate source to output */
}
textComponent &operator|(textComponent &dest)
{
    setOutput(dest);
    return dest;
}
```

This code is used in section [8](#).

10. Each component in the pipeline receives lines through its *get* method, performs whatever processing is in order, then passes them down the pipe to the next component with the *emit* method, which also keeps track of the number of lines generated by this component. For the *textSource* component at the start of the pipeline, this automatically counts lines in the input stream.

Output from a component is normally emitted to the next component in the pipeline, designated by *output*, but may be directed to another component by supplying a pointer to it as the second argument. This permits components to have multiple outputs and hence forks in pipelines. Note that the *lineNumber* in the component counts *all* lines emitted, regardless of the destination.

```

⟨Emit output to next component in pipeline 10⟩ ≡
    virtual void emit(string s, textComponent *destination = Λ)
    {
        if (destination == Λ) {
            destination = output;
        }
        if (destination == Λ) {
            throw (invalid_argument("void_destination_in_emit()"));
        }
        lineNumber++;
        destination->put(s);
    }

```

This code is used in section 8.

11. When the source at the head of the pipeline reaches the end of the input to be processed, it performs whatever end of file processing is appropriate and passes an end of file notification down the pipeline. The default mechanism for handling this is the *eof* method defined in the superclass, which does nothing except forward the notification onward.

If a component needs to perform local cleanup at end of file (for example, if it's buffered look ahead data which needs to be flushed out), it should override the default *eof* method with one which does whatever local processing is needed, then calls *eof* in its parent class to pass the notification down the pipe.

```

⟨Handle end of file notification 11⟩ ≡
    virtual void eof(void)
    {
        output->eof();
    }

```

This code is used in section 8.

12. Source components.

The head end of a filter pipeline must be a *textSource*. It obtains its input from some external source and passes it to the next component in the pipeline. A source drives the pipeline when *send* is called; this reads successive lines from the source and passes them to the next item in the pipeline.

⟨ Class definitions 8 ⟩ +≡

```

class textSource : public textComponent {
protected:
    virtual bool get(string &s) = 0;    /* Get next string from source */
public:
    textSource()
    {
        fType = SourceType;
        source = this;    /* A source is its very own source, of course */
    }
    void put(string s)
    {
        throw (invalid_argument("cannot_put_to_a_source"));
    }
    virtual void send(void)
    {
        /* Send lines from source to next in chain */
        string s;
        while (get(s)) {
            emit(s);
        }
        eof();    /* Notify downstream components of end of file */
    }
};

```

13. Stream source.

The *streamSource* is a source which reads text lines from an input stream. The *setStripEOL* method may be used to set a mode which causes MS-DOS carriage returns left on the ends of lines to be removed.

⟨Class definitions 8⟩ +=

```

class streamSource : public textSource {
private:
    istream *i;
    bool strip;
protected:
    bool get(string &s)
    {
        return getline(*i, s);
    }
public:
    string componentName(void)
    {
        return "streamSource";
    }
    void openFile(string pathName)    /* Bind an input file to the stream source */
    {
        if (pathName == "-") {
            i = &cin;
        }
        else {
            i = new ifstream(pathName.c_str(), ios::in);
            if (!(*i)) {
                throw (invalid_argument("Cannot open input file \" " + pathName + "\"."));
            }
        }
    }
    streamSource(istream &is = cin)
        /* Construct a stream source from an existing input stream */
    {
        i = &is;
        strip = false;
    }
    streamSource(string pathName)
        /* Construct a stream to read from specified pathName; "-" denotes standard input */
    {
        openFile(pathName);
        strip = false;
    }
    void setStripEOL(bool dostrip)
    {
        strip = dostrip;
    }
    bool getStripEOL(void)
    {
        return strip;
    }
}

```

```
virtual void emit(string s, textComponent *destination =  $\Lambda$ )
{
    if (strip) {
        if (s[s.length() - 1]  $\equiv$  '\r') {
            s.erase(s.length() - 1, 1);
        }
    }
    textSource::emit(s, destination);
}
};
```

14. Sink components.

A *textSink* forms the tail of a filter pipeline. It consumes lines from the pipeline and writes them to the ultimate destination.

⟨Class definitions 8⟩ +≡

```
class textSink : public textComponent {
public:
    textSink()
    {
        fType = SinkType;
    }
    void setOutput(textComponent &ofilt)
    {
        throw (invalid_argument("cannot_setOutput_of_a_sink"));
    }
    virtual void put(string s)
        /* Default put method keeps track of lines output to sink destination. */
    {
        lineNumber++;
    }
    virtual void eof(void) /* Default end of file action for a sink is to do nothing, as there's no
                           component downstream to receive the EOF notification. */
    {}
};
```

15. Stream sink.

A *streamSink* writes output sent it to an output stream. Two constructors permit you to create a *streamSink* to write to an already open **ostream** or to a specified file name or standard output.

⟨ Class definitions 8 ⟩ +≡

```

class streamSink : public textSink {
private:
    ostream *o;
public:
    string componentName(void)
    {
        return "streamSink";
    }
    streamSink(ostream &os)
        /* Construct a stream sink that writes to an existing output stream */
    {
        o = &os;
    }
    streamSink(string pathName)
        /* Construct a stream sink that writes to a named pathName; “-” denotes standard output */
    {
        if (pathName ≡ "-") {
            o = &cout;
        }
        else {
            o = new ofstream(pathName.c_str( ), ios::out);
        }
    }
    void put(string s)
    {
        if (&s ≠ Λ) {
            *o << s << "\n";
            textSink::put(s);    /* Call parent to update line counter */
        }
    }
};

```

16. Heat sink.

A *heatSink* discards all data sent to it. As the process of erasing its input is necessarily dissipative; *heatSink* thermalises the information content it receives, increasing the entropy of the universe. See: Bennett, C.H. “The Thermodynamics of Computation—a Review”. *Int. J. Theor. Phys.* **21**:905–940 (1982).

You can use *heatSink* as the final component in a pipeline where the desired output is a side effect of an earlier component, for example, the diagnostic messages produced by *auditFilter*. On a UNIX system you could use **streamSink** with a destination of `/dev/null` for this purpose, but that will not work on other operating systems.

⟨ Class definitions 8 ⟩ +≡

```
class heatSink : public textSink {
public:
    string componentName(void)
    {
        return "heatSink";
    }
    void put(string s)
    {}
};
```

17. Filter components.

Each filter receives its input through its *put* method and delivers output to the next item in the pipeline by calling the *put* method of its designed *output*.

⟨ Class definitions 8 ⟩ +≡

```
class textFilter : public textComponent {
public:
    textFilter()
    {
        fType = FilterType;
    }
};
```

18. Trim filter.

A *trimFilter* removes any blank space from the end of strings which pass through it.

⟨ Class definitions 8 ⟩ +≡

```
class trimFilter : public textFilter {
public:
    string componentName(void)
    {
        return "trimFilter";
    }
    void put(string s)
    {
        while (s.length() > 0 ∧ isspace(*(s.end() - 1))) {
            s.erase(s.end() - 1);
        }
        emit(s);
    }
};
```

19. Tab expander filter.

A *tabExpanderFilter* replaces tab characters with spaces to align to the specified *tabInterval*. We assume tab stops are set at uniform intervals.

⟨Class definitions 8⟩ +≡

```
class tabExpanderFilter : public textFilter {
private:
    int tabInterval;
public:
    string componentName(void)
    {
        return "tabExpanderFilter";
    }
    tabExpanderFilter(int interval = 8)
    {
        setTabInterval(interval);
    }
    void setTabInterval(int interval)
    {
        tabInterval = interval;
    }
    void put(string s)
    {
        if (s.find('\t') ≠ string::npos) {
            ⟨Expand tabs in text line 20⟩;
        }
        assert(s.find('\t') ≡ string::npos);
        emit(s);
    }
};
```

20. Given a string *s* which may contain horizontal tab characters, replace the tabs with spaces to achieve the same alignment, assuming tab stops are set every *tabInterval* columns.

⟨Expand tabs in text line 20⟩ ≡

```
string os;
string::iterator p;
int n = 0;
for (p = s.begin(); p ≠ s.end(); p++) {
    if (*p ≡ '\t') {
        do {
            os += ' ';
            n++;
        } while ((n % tabInterval) ≠ 0);
    }
    else {
        os += *p;
        n++;
    }
}
s = os;
```

This code is used in section 19.

21. Flatten ISO characters filter.

A *flattenISOCharactersFilter* replaces ISO-8859/1 characters with their closest 7-bit ASCII representation. This butchers any text containing accented characters, but if the user asks for it, ya gotta do what ya gotta do.

```

⟨ Class definitions 8 ⟩ +=
class flattenISOCharactersFilter : public textFilter {
public:
    string componentName(void)
    {
        return "flattenISOCharactersFilter";
    }
    void put(string s)
    {
        ⟨ Flatten ISO 8859 characters to 7-bit ASCII 22 ⟩;
        emit(s);
    }
};

```

22. Given a string *s* which may contain ISO-8859/1 characters with codes between #A0–#FF, return a string with all such characters replaced by the closest ASCII equivalents.

```

⟨ Flatten ISO 8859 characters to 7-bit ASCII 22 ⟩ ≡
string os;
string::iterator p;
int c;
for (p = s.begin(); p ≠ s.end(); p++) {
    c = (*p) & #FF;
    if ((c ≥ #A0) ∧ (c ≤ #FF)) {
        os += flattenISO[c - #A0];
    }
    else {
        os += c;
    }
}
s = os;

```

This code is used in section 21.

23. Convert foreign character set to ISO filter.

A *convertForeignCharacterSetToISOFilter* converts characters in a foreign character set to ISO 8859-1. It is driven by a conversion table provided when the filter is instantiated or set by the *setConversionTable* method.

⟨Class definitions 8⟩ +=

```

class convertForeignCharacterSetToISOFilter : public textFilter {
private:
    unsigned char *conversionTable;
public:
    void setConversionTable(unsigned char *tbl)
    {
        conversionTable = tbl;
    }
    convertForeignCharacterSetToISOFilter(unsigned char *tbl)
    {
        setConversionTable(tbl);
    }
    string componentName(void)
    {
        return "convertForeignCharacterSetToISOFilter";
    }
    void identityTransform(void)
    {
        int i;
        conversionTable = new unsigned char[256];
        for (i = 0; i < 256; i++) {
            conversionTable[i] = i;
        }
    }
    unsigned char convert(unsigned char from)
    {
        return conversionTable[from];
    }
    void setTranslation(unsigned char from, unsigned char to)
    {
        conversionTable[from] = to;
    }
    void put(string s)
    {
        string::iterator p;
        for (p = s.begin(); p != s.end(); p++) {
            *p = convert((*p) & #FF);
        }
        emit(s);
    }
};

```

An Etext is divided into three sections, the *prologue*, *body*, and *epilogue*, delimited by the *sectionSep* marker which consists of a 68 character line filled with the sequence `<><><>...<><><>`. The section separator processes lines of the input stream in sequence, testing each against the section separator. Lines prior to the first section separator are emitted to the *prologueProcessor* component, lines within the body to the regular *output* of the component, and lines following the separator at the end of the body (if any) to the *epilogueProcessor* component. If the *prologueProcessor* or *epilogueProcessor* pointers are Λ , output for the corresponding section will be discarded.

[illegible]

```

class sectionSeparatorSquid : public textFilter {
private:
    textComponent *prologueProcessor, *epilogueProcessor, *currentOutput;
    int nsep;
public:
    sectionSeparatorSquid(textComponent *proP =  $\Lambda$ , textComponent *epiP =  $\Lambda$ )
    {
        prologueProcessor = proP;
        epilogueProcessor = epiP;
        nsep = 0;
        currentOutput = prologueProcessor;
    }
    string componentName(void)
    {
        return "sectionSeparatorSquid";
    }
    void setPrologueProcessor(textComponent *proP)
    {
        assert(prologueProcessor  $\equiv \Lambda \wedge$  currentOutput  $\equiv \Lambda$ );
        currentOutput = prologueProcessor = proP;
    }
    void setEpilogueProcessor(textComponent *epiP)
    {
        assert(epilogueProcessor  $\equiv \Lambda$ );
        epilogueProcessor = epiP;
    }
    < Section separator squid end of file handling 26 >;
    void put(string s)
    {
        if (s.compare(sectionSep)  $\equiv$  0) {
            < Handle section separator 25 >;
        }
        if (currentOutput  $\neq \Lambda$ ) {
            emit(s, currentOutput);
        }
    }
};

```

25. The section separator squid is rather flexible in the ways it permits you to direct contents of the sections, and this makes for a modicum of complexity when we see a section separator and wish to redirect the incoming stream. First of all, any of the three output branches—prologue, body, or epilogue—may be discarded by directing them to a Λ component pointer. Further, you may specify the same component as output for more than one branch; for example, if you wish to concatenate the prologue and epilogue into one file.

We need to provide the conventional end of file notification by calling our output components' *eof* methods after they've received the last line of output, but since a component may be attached to more than one branch, when we're switching branches we only want to call *eof* when the component does not appear in a subsequent branch.

```

⟨Handle section separator 25⟩ ≡
  switch (nsep) {
  case 0:
    nsep++;      /* Advance to body */
    if ((currentOutput ≠  $\Lambda$ ) ∧ (currentOutput ≠ output) ∧ (currentOutput ≠ epilogueProcessor)) {
      currentOutput→eof();
    }
    currentOutput = output;      /* Direct output to main component output */
    return;      /* Discard section separator */
  case 1:
    nsep++;
    if ((currentOutput ≠  $\Lambda$ ) ∧ (currentOutput ≠ epilogueProcessor)) {
      currentOutput→eof();
    }
    currentOutput = epilogueProcessor;      /* Direct output to epilogue processor */
    return;      /* Discard section separator */
  case 2:      /* Extra sectionSep in epilogue. Treat as part of epilogue. */
    break;
  }

```

This code is used in section 24.

26. Our much-vaunted “flexibility” in output arrangements also has consequences for end of file processing. When we receive an end of file notification, we can be in any of the three sections, emitting or discarding output, and with potentially identical destinations for sections subsequent to the one which contained the end of file. We thus need to guarantee that not only the current section destination is notified of the end of file (unless it’s Λ), but also that destinations for subsequent sections which will never receive any lines are notified *unless they are the same as the destination for an earlier section which has been notified*.

⟨Section separator squid end of file handling 26⟩ ≡

```
void sectionSeparatorSquid::eof(void)
{
    if (currentOutput ≠  $\Lambda$ ) {      /* Notify current destination unless it's  $\Lambda$  */
        currentOutput→eof();
    }
    switch (nsep) {
    case 0:      /* In prologue. Notify body of eof unless it's  $\Lambda$  or the same destination as prologue. If the
                  epilogue destination is the same as that of the prologue,  $\Lambda$  it out so it isn't notified twice. */
        if ((currentOutput ≠ output) ∧ (output ≠  $\Lambda$ )) {
            output→eof();
            if (epilogueProcessor ≡ currentOutput) {
                epilogueProcessor =  $\Lambda$ ;      /* eof already sent */
            }
        }
        currentOutput = output;
        /* Wheee!!! Fall-through... */
    case 1:      /* End of file encountered in the body. Notify the epilogue destination it's not going to be
                  getting any output unless it's  $\Lambda$  or the same destination as the body, which has already been
                  notified. */
        if ((currentOutput ≠ epilogueProcessor) ∧ (epilogueProcessor ≠  $\Lambda$ )) {
            epilogueProcessor→eof();
        }
    case 2:      /* End of file in the epilogue. No special handling is required. */
        break;
    }
}
```

This code is used in section 24.

27. Tee squid.

The tee squid makes a simple fork in a pipeline. It copies everything it receives to both the component next in the pipeline and the component designated as its *secondDestination*.

⟨ Class definitions 8 ⟩ +≡

```

class teeSquid : public textFilter {
private:
    textComponent *secondDestination;
public:
    teeSquid(textComponent *secP)
    {
        secondDestination = secP;
    }
    string componentName(void)
    {
        return "teeSquid";
    }
    void eof(void)
    {
        secondDestination->eof();
        textFilter::eof();
    }
    void put(string s)
    {
        emit(s, secondDestination);
        emit(s);
    }
};

```

28. Etext body parser filter.

This filter processes the body of an Etext (if the source document contains a prologue and epilogue, this filter should be placed downstream of a **sectionSeparatorSquid**), identifying components in the text and passing them down the pipeline tagged with their type. The body parser is implemented as a state machine, driven by the lines of body copy it receives through its *put* method.

⟨ Class definitions 8 ⟩ +≡

```

class etextBodyParserFilter : public textFilter {
private:
    bodyState state;      /* Current state of parser */
    queue<string> lq;      /* Queue for lines during look-ahead */
    string specialFilter;  /* Filter special commands ? */
    void emits(bodyStates, char bracket, string text = "")
    {
        /* Emit coded line */
        string bracks = "";
        bracks += bracket;
        emit(EncodeBodyState(s) + bracks + text);
    }
    void emitQueuedLines(bodyStates); /* Emit lines in lq with bracketed state s */
public:
    etextBodyParserFilter()
    {
        state = BeginText;
        specialFilter = "";
    }
    virtual ~etextBodyParserFilter()
    {}
    string componentName(void)
    {
        return "etextBodyParserFilter";
    }
    void setSpecialFilter(string f)
    {
        specialFilter = f;
    }
    string getSpecialFilter(void)
    {
        return specialFilter;
    }
    void eof(void)
    {
        emits(EndOfText, Void);
        textFilter::eof();
    }
    void put(string s)
    {
        bodyState lineClass = classifyLine(s);
        if (specialFilter ≠ "") {
            if (isLineSpecial(s)) {
                if (specialType(s) ≠ specialFilter) {

```

```

        return; /* Discard special line not matching filter */
    }
}
}
<Parser state machine 29>;
}

static bodyState classifyLine(string s); /* Classify line by justification type */
static bool isLineSpecial(string s); /* Test for special command */
static string specialType(string s); /* Extract type of special command */
static string specialCommand(string s); /* Extract body of special command */
};

```

29. We enter the parser state machine with two pieces of information: the current *state* of the machine and the *lineClass* of the line just passed to the *put* method. The state machine consists of a **switch** statement with cases for each possible state, wrapped in an endless loop which permits cycling the machine with the same input line after a state change with a simple **continue** statement. This means, of course, that we need to break out of the machine explicitly when we've consumed the input line, but this is simply accomplished with a **break** at the bottom of the loop.

```

<Parser state machine 29> ≡
while (true) {
    switch (state) {
        <BeginText state 30>;
        <BeforeTitle state 31>;
        <Declarations state 32>;
        <PossibleTitle state 33>;
        <TitleMarker state 34>;
        <Author state 35>;
        <BetweenParagraphs state 36>;
        <Within aligned paragraph state 37>;
        <Within preformatted table state 38>;
        <PossibleChapterNumber state 39>;
        <ChapterMarker state 40>;
        <ChapterName state 41>;
        default: cerr << "Internal_error:_state_" << stateNames[state] <<
            "\n_not_handled_in_etextBodyParserFilter.\n";
            exit(1);
    }
    break;
}

```

This code is used in section 28.

30. The state machine starts in *BeginText* state. All we do is emit the corresponding marker to identify the start of the text and drop into *BeforeTitle* state to process the line.

```

<BeginText state 30> ≡
case BeginText: emits(BeginText, Void);
    state = BeforeTitle;
    continue;

```

This code is used in section 29.

31. Once we've output the *BeginText* marker we arrive in this state. At this point we're waiting to encounter either the title and author sequence or the start of document body if no such sequence exists.

⟨BeforeTitle state 31⟩ ≡

```

case BeforeTitle:
  if (lineClass ≠ BetweenParagraphs) {      /* Discard blank lines before title/start of text */
    if (isLineSpecial(s)) {
      emits(Declarations, Begin);
      state = Declarations;
      continue;
    }
    if (lineClass ≡ InCentred) {
      state = PossibleTitle;
      lq.push(s);
      break;
    }
    if (lineClass ≡ TitleMarker) {          /* Weird-title marker with no title */
      state = TitleMarker;                /* Set state to accept author */
      emits(DocumentTitle, Void);        /* Indicate no document title */
      break;
    }
    /* Anything else is start of document with no title or author specified */
    emits(DocumentTitle, Void);
    emits(Author, Void);
    state = BetweenParagraphs;
    continue;    /* Re-parse line in BetweenParagraphs state */
  }
break;

```

This code is used in section 29.

32. One or more format-specific special commands may appear before the document title. These are generally used for document-wide declarations which need to appear before the body of the text. They are returned in a *Declarations* block consisting of all consecutive special commands which appear before the title. If you separate blocks of declarations by blank lines, multiple declarations blocks will be returned; this is generally a dopey thing to do.

⟨Declarations state 32⟩ ≡

```

case Declarations:
  if (isLineSpecial(s)) {
    emits(Declarations, Body, s);
    break;
  }
  emits(Declarations, End);
  state = BeforeTitle;
  continue;

```

This code is used in section 29.

33. We have seen a centred line at the start of the document. This may be a title, or it may simply be centred text which happens to be at the start of a document with no title. We save centred lines in the *lq* queue until we either encounter a line which isn't centred or a title marker.

⟨PossibleTitle state 33⟩ ≡

case *PossibleTitle*:

```
  if (lineClass ≡ TitleMarker) {      /* Title marker—lines saved were the title! */
    emitQueuedLines(DocumentTitle);
    state = TitleMarker;
    break;
  }
```

```
  if (lineClass ≡ InCentred) {
    lq.push(s);      /* Another centred line—save it */
    break;
  }
```

```
  if (lineClass ≡ ChapterMarker) {    /* Chapter marker—it was a chapter! */
    /* We get here if the document doesn't have a title specification but begins with a chapter marker.
       We need to emit a void title and author, then output the centred lines in the queue as a chapter
       number. */
    emits(DocumentTitle, Void);
    emits(Author, Void);
    emitQueuedLines(ChapterNumber);
    state = ChapterMarker;
    break;
  }
```

```
    /* Anything else means the lines in the queue are just centred text at the start of the document.
       Emit a void title and author, then the lines as a centred sequence. */
    emits(DocumentTitle, Void);
    emits(Author, Void);
    emitQueuedLines(InCentred);
    state = BetweenParagraphs;
    continue;
  }
```

This code is used in section 29.

34. We have seen and processed a title marker. Subsequent centred lines are the author specification and will be output as such. The author sequence is terminated by any non-centred line, but blank lines are permitted between the title marker and the first line of the author specification.

⟨TitleMarker state 34⟩ ≡

case *TitleMarker*:

```
  if (lineClass ≡ InCentred) {
    emits(Author, Begin);
    emits(Author, Body, s);
    state = Author;
    break;
  }
```

```
  if (lineClass ≡ BetweenParagraphs) {
    break;      /* Discard blank line after title marker */
  } /* No author specification. Emit void author and process as text */
  emits(Author, Void);
  state = BetweenParagraphs;
  continue;
}
```

This code is used in section 29.

35. One or more lines of the author specification have been output. Successive centred lines are continuation of the author, while anything else ends the author specification.

```

< Author state 35 > ≡
case Author:
  if (lineClass ≡ InCentred) {
    emits(Author, Body, s);
    break;
  }
  emits(Author, End);
  state = BetweenParagraphs;
  continue;

```

This code is used in section 29.

36. *BetweenParagraphs* is the ground state while processing the bulk of the document. Here we have no object open or pending, and we're ignoring blank lines waiting to see something whose alignment determines the handling of the next item we're to process. If it's text (justified, ragged right, ragged left, or block quote), we begin a sequence of that type and set the state to accrue subsequent lines of the same kind. A centred line, however, may be the first line of a chapter break, so we must save it in the *lq* queue and go into *PossibleChapterNumber* state pending examination of the next line.

```

< BetweenParagraphs state 36 > ≡
case BetweenParagraphs:
  switch (lineClass) {
  case BetweenParagraphs:
    break; /* Nugatory blank line */
  case InTextParagraph:
  case InRaggedRight:
  case InBlockQuote:
  case InRaggedLeft:
  case InPreformattedTable:
    emits(lineClass, Begin); /* Emit begin of aligned block */
    emits(lineClass, Body, s);
    state = lineClass;
    break;
  case InCentred: /* Regular centred line */
  case TitleMarker: /* Doesn't belong here, but who knows? */
    lq.push(s);
    state = PossibleChapterNumber;
    break;
  case ChapterMarker: /* Chapter marker without preceding number */
    emits(ChapterNumber, Void);
    state = ChapterMarker;
    break;
  }
  break;

```

This code is used in section 29.

37. This section handles all kinds of aligned paragraphs: justified, ragged left and right, and block quote. As long as we continue to receive lines with same alignment as the state we're in, we simply emit them as continuations of the current paragraph. Upon encountering a line with a different classification, we close the paragraph, revert to *BetweenParagraphs* state, and re-parse the line in that state.

Note how the fact that *classifyLine* uses the same codes for its alignment classes as we use for state when within a paragraph with that alignment pays a big dividend of simplification here.

⟨ Within aligned paragraph state 37 ⟩ ≡

```

case InTextParagraph:
case InRaggedRight:
case InBlockQuote:
case InRaggedLeft:
    if (lineClass ≡ state) {
        emits(state, Body, s);
        break;
    }
    emits(state, End);
    state = BetweenParagraphs;
    continue;

```

This code is used in section 29.

38. To give the maximum latitude for formatting in preformatted tables, once we've identified the first line as beginning in the *PreformattedTableIndent* column and containing at least one sequence of three or more spaces, we stay in preformatted table state until we encounter a blank line.

⟨ Within preformatted table state 38 ⟩ ≡

```

case InPreformattedTable:
    if (lineClass ≠ BetweenParagraphs) {
        emits(state, Body, s);
        break;
    }
    emits(state, End);
    state = BetweenParagraphs;
    break;

```

This code is used in section 29.

39. When we encounter a centred line, there are two possibilities. It may simply be the first of one or more centred lines in the document, or it may be the first line of a chapter break, in which case it belongs to a chapter number specification. We can't distinguish these alternatives until we see either a chapter marker or something other than a line of centred text (including a blank line). As long as we continue to receive centred lines, add them to the *lq* queue. If we get a chapter marker, output the lines in the queue as a chapter number and change state to process the chapter name; otherwise, emit the queued lines as a centred paragraph, reset to *BetweenParagraphs* state, and re-parse the non-centred line.

```

< PossibleChapterNumber state 39 > ≡
case PossibleChapterNumber:
  if (lineClass ≡ InCentred) {
    lq.push(s);
    break;
  }
  if (lineClass ≡ ChapterMarker) {
    emitQueuedLines(ChapterNumber);
    state = ChapterMarker;
    break;
  }
  emitQueuedLines(InCentred);
  state = BetweenParagraphs;
  continue;

```

This code is used in section 29.

40. We've identified a chapter marker and processed the preceding chapter number, if any. Centred lines following the chapter number are output as the chapter name. Any non-centred line, including a blank one, terminates the chapter name. Hence, a chapter marker followed by a blank line denotes a chapter with no title.

```

< ChapterMarker state 40 > ≡
case ChapterMarker:
  if (lineClass ≡ InCentred) {
    emits(ChapterName, Begin);
    emits(ChapterName, Body, s);
    state = ChapterName;
    break;
  }
  emits(ChapterName, Void);
  state = BetweenParagraphs;
  continue;

```

This code is used in section 29.

41. Once we’ve seen a centred chapter name line following a chapter mark, we consider subsequent centred lines as continuations of the chapter name. Anything else (including a blank line) terminates the chapter name and is re-parsed in *BetweenParagraphs* state.

```

⟨ ChapterName state 41 ⟩ ≡
case ChapterName:
  if (lineClass ≡ InCentred) {
    emits(ChapterName, Body, s);
    break;
  }
  emits(ChapterName, End);
  state = BetweenParagraphs;
  continue;

```

This code is used in section 29.

42. The *classifyLine* function examines a line of the body copy and classifies it based on its “heuristic” justification and content, returning a context-free subset of the parser’s *bodyState* values. If you need to modify how the program decides a line should be justified based on how it’s aligned in the input text, here is where you should be looking.

```

⟨ Class definitions 8 ⟩ +≡
bodyState etextBodyParserFilter :: classifyLine(string s)
{
  bodyState classification;
  if (s.length() ≡ 0) {
    classification = BetweenParagraphs;    /* Blank line */
  }
  else if (s[0] ≠ '␣') {
    classification = InTextParagraph;    /* Justified body copy */
  }
  else {
    int i = s.find_first_not_of('␣');
    if (i ≡ RaggedRightIndent) {
      classification = InRaggedRight;    /* Ragged right text */
    }
    else if ((i ≡ PreformattedTableIndent) ∧ (s.find_first_of("␣␣␣") ≠ string::npos)) {
      classification = InPreformattedTable;    /* Preformatted table */
    }
    else if (i ≡ QuoteIndent) {
      classification = InBlockQuote;    /* Block quotation */
    }
    else if (s.length() ≡ FormatWidth) {
      classification = InRaggedLeft;    /* Ragged left */
    }
    else {
      ⟨ Classify centred line 43 ⟩;
    }
  }
}
return classification;
}

```

43. An indented non-blank line which begins in neither the *RaggedRightIndent* nor *QuoteIndent* columns is taken to be centred. We further classify centred lines as either regular text or separators between document title and author lines or chapter number and name specifications which are denoted by centred lines exclusively composed of, respectively, equal (=) or minus (-) signs.

The way in which we recognise these markers looks a little cowboy style unless you realise we already know several important things about the string *s* before we arrive here: it is guaranteed to have no trailing white space, to have at least one blank at the beginning and at least one non-blank thereafter, and to contain no white space characters other than spaces. All of these conditions are guaranteed either by tests within this filter or transformations performed on the input by previous components in the pipeline.

```

⟨ Classify centred line 43 ⟩ ≡
  classification = InCentred;    /* Tentatively classify as centred text */
  char lchar = s[length() - 1];
  if ((lchar ≡ ChapterMarkerCharacter) ∨ (lchar ≡ TitleMarkerCharacter)) {
    int fchar = s.find_first_not_of(' ');
    if (((s.length() - fchar) ≥ MarkerMinimumLength) ∧ (s.find_first_not_of(lchar, fchar) ≡ string::npos))
        {
      classification = (lchar ≡ TitleMarkerCharacter) ? TitleMarker : ChapterMarker;
    }
  }

```

This code is used in section 42.

44. “Special” commands are text lines interpreted by a specific output format generator. Such commands may be used, for example, to include image files in the generated document. Special commands follow the heuristic justification rules of regular text, and are identified by beginning and ending with the *SpecialMarker* sentinel, with the complete *SpecialPrefix* at the start. The *isLineSpecial* function tests whether a line is so marked and should be interpreted as a special command. The function is **public** and **static**, and may be called by downstream components to determine whether a line they have received is a special command.

```

⟨ Class definitions 8 ⟩ +=
  bool etextBodyParserFilter::isLineSpecial(string s)
  {
    unsigned int first = s.find_first_not_of(' ');
    if ((first ≠ string::npos) ∧ (s.find(SpecialPrefix) ≡ first) ∧ (s.rfind(SpecialMarker) ≡
        (s.length() - ((sizeof SpecialMarker) - 1)))) {
      return true;
    }
    return false;
  }

```

45. Each special command contains a type which identifies which output format generators are interested in it. This function, which assumes the line is a special command (*isLineSpecial* returns *true* for it), extracts the type from the command and returns it. This is cowboy code—if you have trouble, try adding an *assert(isLineSpecial(s))* at the top of the function and see if it pops.

⟨Class definitions 8⟩ +=

```
string etextBodyParserFilter :: specialType(string s)
{
    string o = "␣-␣invalid␣-";
    unsigned int first = s.find(SpecialPrefix), last;
    if (first ≠ string::npos) {
        first += (sizeof SpecialPrefix) - 1;
        last = s.find('␣', first);
        o = s.substr(first, last - first);
    }
    return o;
}
```

46. Extract the output format specific body from a special command. If given an invalid special command, an empty string will be returned. *specialCommand* may be called by downstream components to extract the command body from a special command it has received.

⟨Class definitions 8⟩ +=

```
string etextBodyParserFilter :: specialCommand(string s)
{
    string o = "";
    unsigned int first = s.find(SpecialPrefix), last;
    if (first ≠ string::npos) {
        first += (sizeof SpecialPrefix);
        first = s.find('␣', first);
        if (first ≠ string::npos) {
            last = s.rfind(SpecialMarker);
            if (last ≠ string::npos) {
                o = s.substr(first + 1, (last - first) - 1);
            }
        }
    }
    return o;
}
```

47. This little helper function emits lines stored in the look-ahead queue *lq* as a block of lines of a given type *s*, complete with *Begin* and *End* brackets.

⟨Class definitions 8⟩ +=

```
void etextBodyParserFilter :: emitQueuedLines(bodyStates)
{
    emits(s, Begin);
    while (¬lq.empty()) {
        emits(s, Body, lq.front());
        lq.pop();
    }
    emits(s, End);
}
```

48. These are the states among which the Etext body parser transitions as it processes lines of the text. Note that some of these states are also used by *classifyLine* to denote the justification of individual lines.

```
#define EncodeBodyState(s) ((char)('A' + (s)))
#define DecodeBodyState(c) ((bodyState)((c) - 'A'))

⟨ Global variables 48 ⟩ ≡
enum bodyState {          /* Body parser current state */
    BeginText,           /* Begin text pseudo-marker */
    BeforeTitle,         /* Title not yet seen */
    Declarations,        /* Special declarations before title */
    PossibleTitle,        /* Centred text which may be the title */
    TitleMarker,         /* Separator between title and author */
    DocumentTitle,       /* Document title */
    Author,              /* Author information after title separator */
    BetweenParagraphs,    /* Blank space between paragraphs */
    InTextParagraph,     /* In regular text paragraph */
    InBlockQuote,        /* In indented block quotation paragraph */
    InRaggedRight,       /* In ragged right paragraph */
    InRaggedLeft,        /* In ragged left paragraph */
    InPreformattedTable, /* In preformatted table */
    PossibleChapterNumber, /* Centred text which may be chapter number */
    InCentred,           /* In centred text */
    ChapterNumber,       /* Chapter number */
    ChapterMarker,       /* Marker after chapter number */
    ChapterName,         /* Chapter name */
    EndOfParagraph,      /* End of paragraph pseudo-marker */
    EndOfText            /* End of text pseudo-marker */
};
```

See also sections 49, 50, 146, 149, 154, 155, and 156.

This code is used in section 6.

49. Each of the syntactic elements recognised by the parser are output to the component downstream with brackets which mark the beginning, body, and end of each element. The *Begin* and *End* markers are sent with the state code identifying the element but no text. If an element such as the title or author is omitted, a *Void* record is output to indicate its absence.

```
⟨ Global variables 48 ⟩ +=
static const char Begin = '{',      /* Structure nesting flags */
    Body = '␣', End = '}', Void = '-';
```

50. For debugging, it's nice to be able to dump the parser states (particularly those with which lines emitted by the parser are tagged). Strings in the following table correspond to the states in **bodyState** and are used by the *parserDiagnosticFilter* to generate its output.

```
⟨ Global variables 48 ⟩ +=
static string const stateNames[] = {"Begin␣text", "B4␣Title", "Declarations", "Poss␣Title",
    "Title␣mark", "Title", "Author", "Par␣break", "Text", "Blockquote", "Rag␣right",
    "Rag␣left", "Table", "Poss␣Chap", "Centred", "Chap␣num", "Chap␣mark", "Chap␣name",
    "End␣para", "End␣text"};
```

51. Strip special commands filter.

This filter scans its input for special commands (identified by the `etextBodyParserFilter::isLineSpecial` function) and removes them from the stream passed down the pipeline. If removal of special commands would result in two consecutive blank lines in the output, the extra blank line is also elided. This filter assumes that its input contains no tab characters nor trailing white space (and hence that any blank line is a zero length string).

⟨ Class definitions 8 ⟩ +≡

```
class stripSpecialCommandsFilter : public textFilter {
private:
    bool lastBlank, lastStripped;
public:
    stripSpecialCommandsFilter()
    {
        lastBlank = lastStripped = false;
    }
    string componentName(void)
    {
        return "stripSpecialCommandsFilter";
    }
    void put(string s)
    {
        if (etextBodyParserFilter::isLineSpecial(s)) {
            lastStripped = true;
        }
        else {
            if (s.length() > 0) {
                emit(s);
                lastStripped = lastBlank = false;
            }
            else {
                if (lastStripped) {
                    if (!lastBlank) {
                        emit(s);
                    }
                    lastStripped = false;
                    lastBlank = true;
                }
                else {
                    emit(s);
                    lastBlank = true;
                }
            }
        }
    }
};
```

52. Audit filter.

The *auditFilter* performs a variety of tests on lines which pass through it. The tests are selected by a bit mask of *audit_criteria* passed to the constructor or set by *setAuditCriteria* (the default is to enable all tests). The *auditFilter* passes input unchanged to the component downstream. Error messages are written to an **ostream** log which defaults to *cerr*. For complete generality, this should probably be replaced with a **textComponent** transforming *auditFilter* into a squid.

⟨ Class definitions 8 ⟩ +≡

```

class auditFilter : public textFilter {
public:
    enum audit_criteria {
        trailing_blanks = 1, embedded_tabs = 2, exceeds_maximum_length = 4, invalid_characters = 8,
        dubious_justification = 16, improper_embedded_blanks = 32, consecutive_blank_lines = 64,
        special_commands_present = 128, permit_8_bit_ISO_characters = 256, trailing_hyphen = 512,
        everything = ~0
    };
private:
    static const int DefaultCentringTolerance = 2;    /* If you're picky, you can set this to 1 */
    unsigned int maxLineLength;
    ostream *log;
    bool lastBlank;
    bool inTable;
    enum audit_criteria check;
    int centringTolerance;
public:
    string componentName(void)
    {
        return "auditFilter";
    }
    void setMaxLength(unsigned int maxlen)
    {
        maxLineLength = maxlen;
    }
    void setLogStream(ostream &s)
    {
        log = &s;
    }
    void setAuditCriteria(int check_for)
    {
        check = (audit_criteria) check_for;
    }
    audit_criteria getAuditCriteria(void)
    {
        return check;
    }
    void enableAuditCriteria(int check_for)
    {
        check = (audit_criteria)(check | check_for);
    }
    void disableAuditCriteria(int check_for)
    {

```

```

    check = (audit_criteria)(check & (~check_for));
}
void setCentringTolerance(int ct = DefaultCentringTolerance)
{
    centringTolerance = ct;
}
int getCentringTolerance(void)
{
    return centringTolerance;
}
auditFilter(unsigned int maxlen = FormatWidth, ostream &os = cerr, audit_criteria
    check_for = everything)
{
    setMaxLength(maxlen);
    setLogStream(os);
    lastBlank = false;
    inTable = false;
    setAuditCriteria(check_for);
    setCentringTolerance();
}
static bool isCharacterPermissible(unsigned int c);
static string quoteArbitraryString(string s);
static bool isISOletter(int c)
{
    assert((c ≥ 0) ∧ (c ≤ #FF));
    return ((c ≥ 'A') ∧ (c ≤ 'Z')) ∨ ((c ≥ 'a') ∧ (c ≤ 'z')) ∨ ((c ≥ #C0) ∧ (c ≤ #D6)) ∨ ((c ≥
        #D8) ∧ (c ≤ #F6)) ∨ (c ≥ #F8);
}
void put(string s)
{
    unsigned int i, n;
    bool err = false;
    const string sentenceEnd = ".?!\"'\"";
    bodyState lclass;
    bool special = etextBodyParserFilter::isLineSpecial(s);
    < Check for line with trailing white space 53 >;
    < Check for line with trailing hyphen 54 >;
    < Check for line with embedded tab characters 55 >;
    < Check for line that exceeds maximum text length 57 >;
    < Check for invalid characters in text 58 >;
    < Check for justification-related problems 59 >;
    < Check for line with improper embedded white space 56 >;
    < Check for consecutive blank lines 60 >;
    < Check for special commands present 61 >;
    if (err) {
        issueMessage(quoteArbitraryString(s), *log);
    }
    emit(s);
}
};

```

53. In the interest of visual fidelity as well as minimising file size, we don't want to include any lines with nugatory white space between the last printable character and the end of line. If any have crept in, generate a warning.

⟨ Check for line with trailing white space 53 ⟩ ≡

```

if (check & trailing_blanks) {
  int j;
  n = 0;
  for (j = s.length() - 1; j ≥ 0; j--) {
    if ( $\neg$ isspace(s[j])) {
      break;
    }
    n++;
  }
  if (n > 0) {
    ostringstream em;
    em << "Line_contains_" << n << "_white_space_character" << (n ≡ 1 ? "" : "s") <<
      "_at_the_end.";
    issueMessage(em.str(), *log);
    err = true;
  }
}

```

This code is cited in section 158.

This code is used in section 52.

54. One common problem in scanned documents is hyphenated lines which were not joined in the editing phase. This check attempts to detect such lines. We only issue the warning if the character that precedes the hyphen is alphabetic (including ISO accented letters), as trailing em-dashes and minus signs in mathematics are perfectly valid.

⟨ Check for line with trailing hyphen 54 ⟩ ≡

```

if (check & trailing_hyphen) {
  if ((s.length() ≥ 2) ∧ (s[s.length() - 1] ≡ '-')) {
    int p = s[s.length() - 2] & #FF;
    if (isISOletter(p)) {
      ostringstream em;
      em << "Line_contains_an_apparent_hyphen_at_the_end.";
      issueMessage(em.str(), *log);
      err = true;
    }
  }
}

```

This code is used in section 52.

55. By the time we audit the text, any tab characters which may have appeared in the input should have been expanded to spaces. Tab characters presume tab stop settings which, while usually defaulting to 8 characters, are nowhere specified in a standard. Leaving tabs in an Etext runs the risk that carefully-aligned material may be scrambled if viewed on a system with different tab stops. Here we verify that no tabs remain. Note that a tab will also fail (Check for invalid characters in text 58), but making a special check here makes the diagnostic for this common case more comprehensible.

(Check for line with embedded tab characters 55) \equiv

```

if (check & embedded_tabs) {
  if ((i = s.find('\\t'))  $\neq$  string::npos) {
    for ( ; i < s.length(); i++) {
      if (s[i]  $\equiv$  '\\t') {
        ostringstream em;
        em << "Line_\\contains_\\tab_\\character_\\in_\\column_" << (i + 1) << ".";
        issueMessage(em.str(), *log);
        err = true;
      }
    }
  }
}

```

This code is used in section 52.

56. Lines should not have more than one space between words except after sentence-ending punctuation. Such lines may result from attempts to justify text by adding space, and may be propagated even if the text is re-aligned. Within a preformatted table embedded spaces are allowed and this test is skipped.

```

⟨ Check for line with improper embedded white space 56 ⟩ ≡
  if ((check & improper_embedded_blanks) ∧ (¬inTable) ∧ (¬special)) {
    i = s.find_first_not_of(' ');    /* Ignore leading spaces */
    if (i ≠ string::npos) {
      while ((i = s.find(" ", i)) ≠ string::npos) {
        if ((i > 0) ∧ (sentenceEnd.find(s[i - 1]) ≠ string::npos)) {
          if (s.substr(i + 2, 1) ≡ " ") {
            ostringstream em;
            em << "Line contains extra embedded space after sentence end in column " <<
              (i + 1) << ".";
            issueMessage(em.str(), *log);
            err = true;
            i += 3;
          }
          else {
            i += 2;
          }
        }
      }
    }
    else {
      ostringstream em;
      em << "Line contains extra embedded space in column " << (i + 1) << ".";
      issueMessage(em.str(), *log);
      err = true;
      i += 2;
    }
  }
}

```

This code is used in section 52.

57. In the interest of human readability we restrict the maximum length of text lines in the document to *maxLineLength* characters. If the line exceeds that limit, issue a warning. Note that we've already tested for lines which still contain embedded tab characters or trailing white space at this point. Special commands are exempted from this check.

```

⟨ Check for line that exceeds maximum text length 57 ⟩ ≡
  if ((check & exceeds_maximum_length) ∧ (¬special) ∧ (s.length() > maxLineLength)) {
    ostringstream em;
    em << "Line (length " << s.length() << ") exceeds maximum of " << maxLineLength <<
      " characters.";
    issueMessage(em.str(), *log);
    err = true;
  }
}

```

This code is used in section 52.

58. Scan the text line to ensure it contains no impermissible characters as defined by *isCharacterPermissible*.[■] One little detail: if we’re explicitly checking for *embedded_tabs*, there’s no need to report them a second time as invalid characters. Finally, if *permit_8_bit_ISO_characters* is not set, we require the input to consist of exclusively 7-bit ASCII characters; ISO characters are reported as errors in this mode.

⟨ Check for invalid characters in text 58 ⟩ ≡

```

if (check & invalid_characters) {
  for (i = 0; i < s.length(); i++) {
    if ((¬isCharacterPermissible(s[i])) ∨ (((¬(check & permit_8_bit_ISO_characters)) ∧ ((s[i] & #FF) ≥ 127))))
      {
        if ((s[i] ≠ '\t') ∨ (¬(check & embedded_tabs))) {
          ostream em;
          em << "Invalid_character_0x" << hex << (s[i] & #FF) << dec << "_in_column_" << (i+1) << ".";
          issueMessage(em.str(), *log);
          err = true;
        }
      }
  }
}

```

This code is cited in section 55.

This code is used in section 52.

59. Use the “heuristic justification” classifier of the **etextBodyParserFilter** to evaluate the line, then verify if the actual content of the line is consistent with its evaluation. We also keep track of whether we’re currently in a preformatted table, within which embedded spaces are permitted. The test for “dubious centred lines” catches a multitude of sins, in particular ragged right, block quote, and ragged left lines which do not start or end in the prescribed columns. Special commands are exempted from this check.

⟨ Check for justification-related problems 59 ⟩ ≡

```

if ((check & dubious_justification) ∧ ¬special) {
  lclass = etextBodyParserFilter::classifyLine(s);
  if (lclass ≡ InPreformattedTable) {
    inTable = true;
  }
  else if (lclass ≡ BetweenParagraphs) {
    inTable = false; /* Only blank line ends table */
  }
  else if (¬inTable ∧ (lclass ≡ InCentred)) {
    int l, r;
    l = s.find_first_not_of(' '); /* Number of leading spaces */
    r = maxLineLength - s.length(); /* Number of (virtual) trailing spaces */
    if (labs(l - r) > centringTolerance) {
      ostream em;
      em << "Dubious_centred_line_" << l << "spaces_at_left_" << r << "spaces_at_right.";
      issueMessage(em.str(), *log);
      err = true;
    }
  }
}

```

This code is used in section 52.

60. There’s no reason for more than one consecutive blank line to appear in the text. Multiple consecutive blank lines are most likely editing errors which would render the raw text less readable.

```

⟨ Check for consecutive blank lines 60 ⟩ ≡
  if (check & consecutive_blank_lines) {
    if (s.find_first_not_of(' ') == string::npos) {
      if (lastBlank) {
        issueMessage("This_and_previous_line_are_both_blank.", *log);
        err = true;
      }
      lastBlank = true;
    }
    else {
      lastBlank = false;
    }
  }
}

```

This code is used in section 52.

61. Output format specific **Special** commands are included in Etexts to facilitate the production of published editions in various formats and media, but should be removed prior to distribution of an Etext in “Plain ASCII” form. (This can be accomplished by passing the Etext through the **stripSpecialCommandsFilter**.) This test detects specials inadvertently left in an Etext intended for publication.

```

⟨ Check for special commands present 61 ⟩ ≡
  if ((check & special_commands_present) & special) {
    issueMessage("Special_command_present_in_text.", *log);
    err = true;
  }
}

```

This code is used in section 52.

62. The body of an Etext must contain nothing other than the ISO-8859/1 printable characters, blanks, and end of line delimiters. You’d think this wouldn’t be a problem, but thanks to Microsoft’s little collection of incompatible horrors jammed right in the middle of the ISO (and Unicode) 8 bit control set, plus editors who amuse themselves by jamming form feeds, vertical tabs, etc. into documents, it pays to be sure, since treating any such nonsense as legitimate text characters may lead to disaster downstream.

The following static helper function determines if its character argument is permissible in Etext body copy. At the time this function is called we assume that any trailing white space including end of line sequences has been deleted and that horizontal tabs have been expanded to spaces. Placing **trimFilter** and **tabExpanderFilter** in the pipeline before **auditFilter** will guarantee these criteria are met.

```

⟨ Class definitions 8 ⟩ +=
  bool auditFilter::isCharacterPermissible(unsigned int c)
  {
    if (c < ' ') {
      return false; /* ASCII control characters not permitted */
    }
    if (c ≥ 127 & c < 161) {
      return false; /* DEL, ISO control characters, or non-breaking space prohibited */
    }
    return true;
  }
}

```

63. When issuing an error message for a string which may contain invalid and/or non-printing characters, we need to quote those characters so they're apparent. This function takes a string containing arbitrary 8 bit characters and returns a string in which all characters other than ASCII and ISO graphics are quoted as C hexadecimal escapes.

⟨Class definitions 8⟩ +=

```
string auditFilter :: quoteArbitraryString(string s)
{
    string o = "";
    string::iterator cp;
    unsigned int c;
    for (cp = s.begin(); cp < s.end(); cp++) {
        c = (*cp) & #FF;
        if (isCharacterPermissible(c)) {
            if ((c == '\u') & (s.find_first_not_of('\u', (cp + 1) - s.begin()) == string::npos)) {
                o += "\\x20";
            }
            else {
                o += c;
            }
        }
        else {
            ostringstream eh;
            eh << "\\x" << hex << setw(2) << setfill('0') << c;
            o += eh.str();
        }
    }
    return o;
}
```

64. Parser diagnostic filter.

This filter processes the body of an Etext (if the source document contains a prologue and epilogue, this filter should be placed downstream of a **sectionSeparatorSquid**), identifying components in the text and passing them down the pipeline tagged with their type. The body parser is implemented as a state machine, driven by the lines of body copy it receives through its *put* method.

⟨Class definitions 8⟩ +=

```
class parserDiagnosticFilter : public textFilter {
private:
public:
    string componentName(void)
    {
        return "parserDiagnosticFilter";
    }
    void put(string s)
    {
        bodyState rtype = DecodeBodyState(s[0]);
        string spaces = "          ", stateName = "";
        stateName += stateNames[rtype];
        emit(s.substr(1,1) + "\u" + stateName + spaces.substr(0,12 - stateName.length()) + s.substr(2));
    }
};
```

65. Utilities.

The following are not full-fledged pipeline components, but rather utilities which provide services to text processing components.

66. Text substituter.

The *textSubstituter* performs replacement of substrings in text with defined substitutes.

⟨Class definitions 8⟩ +≡

```
class textSubstituter {
private:
    deque<string> fromString;
    deque<string> toString;
public:
    void addSubstitution(string from, string to)
    {
        fromString.push_back(from);
        toString.push_back(to);
    }
    string substitute(string s);
};
```

67. The *substitute* method applies all of the substitution rules of the **textSubstituter** to its argument string and returns the result. Note that substitutions are not re-scanned, and hence cannot result in infinite expansion loops.

⟨Class definitions 8⟩ +≡

```
string textSubstituter::substitute(string s)
{
    deque<string>::iterator f = fromString.begin();
    deque<string>::iterator t = toString.begin();
    string o = s;
    while (f != fromString.end()) {
        unsigned int i = 0, n;
        while ((n = o.find(*f, i)) != string::npos) {
            o.replace(n, f-length(), *t);
            i = n + t-length();
        }
        f++;
        t++;
    }
    return o;
}
```

68. LaTeX Generation.

This filter translates parsed body copy (emitted by *etextBodyParser*) into L^AT_EX source code, which it passes down the pipeline.

⟨ Class definitions 8 ⟩ +≡

```

class LaTeXGenerationFilter : public textFilter {
private:
    bool italics, inmath, quoth, hastitle, hasauthor, intable, firstchap;
    int footnest;
    textSubstituter transformer;
    string quoteLaTeXString(string s);
    void emitq(string s)
    {
        emit(quoteLaTeXString(s));
    }
    void generateAlignedParagraph(string envtype, char bracket, string text, string
        terminator = "\\");
    bool isSubstitution(string s);
public:
    LaTeXGenerationFilter()
    {
        italics = inmath = quoth = false;
        hastitle = hasauthor = false;
        intable = firstchap = false;
        footnest = 0;
    }
    virtual ~LaTeXGenerationFilter()
    {}
    string componentName(void)
    {
        return "LaTeXGenerationFilter";
    }
    void put(string s);
};

```

69. The *put* method of the `LaTeXGenerationFilter` wraps `LATEX` commands around the line-level structure of the text to achieve the desired formatting. Since almost all of the real work is done upstream (by `etextBodyParserFilter`) and downstream (by *quoteLaTeXString*) there is actually little that needs doing here.

⟨ Class definitions 8 ⟩ +≡

```
void LaTeXGenerationFilter::put(string s)
{
    bodyState state = DecodeBodyState(s[0]);
    char bracket = s[1];
    string text = s.substr(2);
    switch (state) {
    case BeginText: ⟨ Generate start of document in LaTeX 70 ⟩;
    case Declarations: ⟨ Process declarations in LaTeX 71 ⟩;
    case DocumentTitle: ⟨ Process document title in LaTeX 72 ⟩;
    case Author: ⟨ Process author in LaTeX 73 ⟩;
    case ChapterNumber: break; /* Chapter numbers are totally ignored */
    case ChapterName: ⟨ Process chapter name in LaTeX 74 ⟩;
    case InTextParagraph: ⟨ Generate justified text paragraph in LaTeX 75 ⟩;
    case InBlockQuote: generateAlignedParagraph("quote", bracket, text, "");
        break;
    case InRaggedRight: generateAlignedParagraph("flushleft", bracket, text);
        break;
    case InRaggedLeft: generateAlignedParagraph("flushright", bracket, text);
        break;
    case InPreformattedTable:
        if (bracket == Begin) {
            intable = true;
        }
        generateAlignedParagraph("verbatim", bracket, text, "");
        if (bracket == End) {
            intable = false;
        }
        break;
    case InCentred: generateAlignedParagraph("center", bracket, text);
        break;
    case EndOfText: emit("\\end{document}");
        if (verbose) {
            cerr << "LaTeX:␣" << getLineNumber() << "␣lines␣output.␣n";
        }
        break;
    default: cerr << "***␣State␣" << stateNames[state] << "␣" << bracket <<
        "␣not␣handled␣in␣LaTeXGenerationFilter␣***␣n";
        exit(1);
    }
}
```

70. Generate the boilerplate at the start of a \LaTeX document and declarations appropriate for the type of document we're about to write.

```

⟨Generate start of document in LaTeX 70⟩ ≡
  emit("\documentclass{book}");
  {
    time_t t = time(&);
    string stime = ctime(&t);
    stime = stime.substr(0, stime.length() - 1);
    emit("%Translated by PRODUCT VERSION ("REVDATE") on" + stime);
  }
  if (babelon) {
    emit("\usepackage[" + babelang + "]{babel}");
    emit("\usepackage[latin1]{inputenc}");
    emit("\usepackage[T1]{fontenc}");
  }
  else {
    if (frenchPunct) {
      emit("\frenchspacing");
    }
  }
  break;

```

This code is used in section 69.

71. Any \LaTeX -specific special commands appearing in declaration blocks before the title are output in the preamble of the document where they may load packages and/or make definitions in global scope.

```

⟨Process declarations in LaTeX 71⟩ ≡
  if (bracket ≡ Body) {
    assert(etextBodyParserFilter::isLineSpecial(text));
    if (¬isSubstitution(text)) {
      emit(etextBodyParserFilter::specialCommand(text));
    }
  }
  break;

```

This code is used in section 69.

72. For the document title we need only wrap it in a \LaTeX `title` declaration in the preamble and set `hastitle` to remind us to emit a `maketitle` command at the start of the document body.

```

⟨Process document title in LaTeX 72⟩ ≡
  switch (bracket) {
  case Begin: emit("\title{");
    break;
  case Body: emitq(text);
    break;
  case End: emit("}");
    hastitle = true;
    break;
  case Void: hastitle = false;
    break;
  }
  break;

```

This code is used in section 69.

73. Similarly, the author is wrapped in a \LaTeX `author` declaration in the preamble. When we see the *End* bracket for the author specification (or the *Void* bracket if no author is given), it's time to close the preamble and begin the body of the document. If either a title or author were specified, we then need to emit a `\maketitle` command to create the title. Since `\maketitle` requires both a title and author declaration, if either is missing we supply a blank one before closing the preamble.

```

⟨Process author in LaTeX 73⟩ ≡
  switch (bracket) {
  case Begin: emit("\\author{");
    break;
  case Body: emitq(text);
    break;
  case End: emit("}");
    hasauthor = true;
    /* Note fall-through */
  case Void:
    if (hastitle ∨ hasauthor) {
      if (¬hastitle) {
        emit("\\title{}");
      }
      if (¬hasauthor) {
        emit("\\author{}");
      }
    }
    emit("\\begin{document}");
    if (hastitle ∨ hasauthor) {
      emit("\\maketitle");
    }
    break;
  }
  break;

```

This code is used in section 69.

74. Chapter names cause a `chapter` command to be generated with the chapter title as its argument. *Void* chapter names generate chapters with blank titles.

⟨Process chapter name in LaTeX 74⟩ ≡

```
switch (bracket) {
case Begin:
  if (¬firstchap) {
    firstchap = true;
    emit("\\tableofcontents");
  }
  emit("\\chapter{");
  break;
case Body: emitq(text);
  break;
case End: emit("}");
  break;
case Void: emit("\\chapter{}");
  break;
}
break;
```

This code is used in section 69.

75. Regular justified text paragraphs don't need any special wrapper, as they're the default in L^AT_EX. We output a blank line before the first line of the body to break the previous paragraph and reset the quote parity so the first ASCII quote in the paragraph will be turned into an open quote.

⟨Generate justified text paragraph in LaTeX 75⟩ ≡

```
switch (bracket) {
case Begin: emit("");
  quoth = false;
  break;
case Body:
  if (etextBodyParserFilter::isLineSpecial(text)) {
    if (¬isSubstitution(text)) {
      emit(etextBodyParserFilter::specialCommand(text));
    }
  }
  else {
    emitq(text);
  }
  break;
case End: case Void: break;
}
break;
```

This code is used in section 69.

76. This function handles the various kinds of aligned paragraphs we encounter in a document. It wraps the contents of the paragraph in a \LaTeX environment of the type specified by *envtype*.

⟨ Class definitions 8 ⟩ +≡

```

void LaTeXGenerationFilter::generateAlignedParagraph(string envtype, char bracket, string
    text, string terminator = "\\")
{
    switch (bracket) {
    case Begin: emit("");
        emit("\\begin{" + envtype + "}");
        quoth = false;
        break;
    case Body:
        if (etextBodyParserFilter::isLineSpecial(text)) {
            if (¬isSubstitution(text)) {
                emit(etextBodyParserFilter::specialCommand(text));
            }
        }
        else {
            emit(quoteLaTeXString(text) + terminator);
        }
        break;
    case End: emit("\\end{" + envtype + "}");
        break;
    case Void: break;
    }
}

```

77. Translate text string s into \LaTeX , quoting metacharacters and expanding Latin-1 characters as required. This is where we handle switching to and from math mode, toggling italic, expanding ellipses and em-dashes, and recognising footnotes.

⟨ Class definitions 8 ⟩ +=

```

string LaTeXGenerationFilter::quoteLaTeXString(string s)
{
    string::iterator cp;
    string o = "";
    int c;
    static const string mathModeQuoted = "|<>", punctuation = "?!,:";
    /* Punctuation set after space for frenchPunct */
    quotedCharacters = "$\&\%", quotedTextCharacters = "{}";
    for (cp = s.begin(); cp < s.end(); cp++) {
        c = (*cp) & #FF;
        if (c < '␣') {
            ⟨ Quote control character in LaTeX 78 ⟩;
        }
        else if (c > 160) {
            ⟨ Translate ISO graphic character in LaTeX 79 ⟩;
        }
        else if (c ≥ '␣' ∧ c ≤ '~') {
            if (¬inmath ∧ ¬intable ∧ c ≡ '␣') {
                ⟨ Toggle italic text mode in LaTeX 80 ⟩;
            }
            else if (¬intable ∧ c ≡ '\\') ∧ ((cp + 1) < s.end()) ∧ ((cp[1] ≡ '(') ∨ (cp[1] ≡ ')')) {
                ⟨ Toggle math mode in LaTeX 81 ⟩;
            }
            else if (¬inmath ∧ ¬intable ∧ ((cp + 2) < s.end()) ∧ ((c ≡ '[') ∨ ((c ≡ '␣') ∧ (cp[1] ≡ '[') ∨ ((c ≡ '␣') ∧ (cp[1] ≡ '[') ∧ (cp[2] ≡ '[')))) {
                ⟨ Begin footnote in LaTeX 82 ⟩;
            }
            else if (¬inmath ∧ ¬intable ∧ c ≡ ']') {
                ⟨ End footnote in LaTeX 83 ⟩;
            }
            else if (¬inmath ∧ ¬intable ∧ (c ≡ '-') ∧ ((cp + 1) < s.end()) ∧ (cp[1] ≡ '-')) {
                ⟨ Translate em-dash in LaTeX 84 ⟩;
            }
            else if (¬inmath ∧ ¬intable ∧ (c ≡ '.') ∧ ((cp + 2) < s.end()) ∧ (cp[1] ≡ '.') ∧ (cp[2] ≡ '.')) {
                ⟨ Translate ellipsis in LaTeX 85 ⟩;
            }
            else if (¬intable ∧ ((c ≡ '~') ∨ ((¬inmath) ∧ (c ≡ '^')))) {
                ⟨ Quote ASCII character as verbatim in LaTeX 86 ⟩;
            }
            else if ((¬inmath ∧ ¬intable) ∧ (mathModeQuoted.find_first_of(c) ≠ string::npos)) {
                ⟨ Quote character as math mode in LaTeX 87 ⟩;
            }
            else if (¬inmath ∧ ¬intable ∧ c ≡ '"') {
                ⟨ Convert ASCII quotes to open and close quotes in LaTeX 88 ⟩;
            }
            else {
                ⟨ Output ASCII text character in LaTeX 89 ⟩;
            }
        }
    }
}

```

```

    } /* Note that other characters, specifically those in the range from 127 through 160, get
        dropped. */
}
o = transformer.substitute(o); /* Apply substitutions, if any */
return o;
}

```

78. This is a control character. Emit as $\text{\textasciitilde letter}$ unless it is considered as white space (for example, carriage return and line feed), in which case it's sent directly to the output.

⟨Quote control character in LaTeX 78⟩ \equiv

```

if (isspace(c)) {
    o += c;
}
else {
    o += "\\verb+^";
    o += ('@' + c);
    o += '+';
}

```

This code is used in section 77.

79. This is a graphic character belonging to the ISO 8859 set with character codes between #A0 and #FF. Translate it into the best L^AT_EX equivalent or pass it on to be handled by the babel package if babelon is set.

⟨Translate ISO graphic character in LaTeX 79⟩ \equiv

```

if (babelon) {
    o += c;
}
else {
    o += texform[c - 161];
}

```

This code is used in section 77.

80. The underscore character, “_”, toggles text between the normal roman and italic fonts.

⟨Toggle italic text mode in LaTeX 80⟩ \equiv

```

italics = ¬italics;
if (italics) {
    o += "\\it_";
}
else {
    o += "}";
}

```

This code is used in section 77.

81. We use the same sequences as L^AT_EX, “\(" and “\)”, to toggle between text and math mode, so we can simply emit the sequence unmodified. We need to take note of it, however, to keep track of whether we're in math mode as that affects handling of some other sequences.

⟨Toggle math mode in LaTeX 81⟩ \equiv

```

o += c;
inmath = cp[1] ≡ '(';

```

This code is used in section 77.

82. Footnotes appear in-line, within [square brackets]. Translate them to a \LaTeX footnote environment. Footnotes aren't supposed to be nested in Etexts, but \LaTeX handles them just fine.

```

⟨Begin footnote in LaTeX 82⟩ ≡
  footnest++;
  o += "\\footnote{";
  if ((c ≡ '␣') ∧ ((cp + 1) < s.end())) {
    if (cp[1] ≡ '␣') {
      cp++;
    }
    cp++;
  }
}

```

This code is used in section 77.

83. Close a footnote when the right bracket is encountered.

```

⟨End footnote in LaTeX 83⟩ ≡
  o += '}' ;
  if (footnest ≡ 0) {
    issueMessage("Mismatched␣end␣of␣footnote␣(␣]␣)␣bracket.");
  }
  else {
    footnest--;
  }
}

```

This code is used in section 77.

84. Two adjacent hyphens, “--” denote an *em* dash in an ASCII Etext. Translate this sequence into three hyphens as used by \LaTeX .

```

⟨Translate em-dash in LaTeX 84⟩ ≡
  o += "---";
  cp++;

```

This code is used in section 77.

85. Three consecutive periods are translated into a `\ldots` ellipsis.

```

⟨Translate ellipsis in LaTeX 85⟩ ≡
  o += "\\ldots";
  if (cp[2] ≡ '␣') {
    o += "␣";
  }
  else {
    o += "\\␣";
  }
  cp += 2;

```

This code is used in section 77.

86. This code handles tilde and circumflex characters (the latter only when not in math mode), which must be quoted in a `\verb` sequence to appear in text.

```

⟨Quote ASCII character as verbatim in LaTeX 86⟩ ≡
  o += "\\verb+";
  o += c;
  o += '+' ;

```

This code is used in section 77.

87. The greater, less, and vertical bar symbols cannot appear in regular text in \LaTeX ; output them in math mode.

\langle Quote character as math mode in \LaTeX 87 $\rangle \equiv$

```
o += '$';
o += c;
o += '$';
```

This code is used in section 77.

88. ASCII quote characters are translated into open and close quote symbols. Note that the flag *quoth* is unconditionally reset at the end of a paragraph so mismatched quotes won't propagate beyond one paragraph. This allows continued quotes in multiple paragraphs to work properly.

\langle Convert ASCII quotes to open and close quotes in \LaTeX 88 $\rangle \equiv$

```
o += quoth ? "''" : "''";
quoth = ¬quoth;
```

This code is used in section 77.

89. Output a text character. Some \LaTeX metacharacters require backslash quoting in any mode, others only outside math mode.

\langle Output ASCII text character in \LaTeX 89 $\rangle \equiv$

```
if (¬inmath ∧ frenchPunct ∧ (punctuation.find_first_of(c) ≠ string::npos) ∧ (((cp + 1) ≡ s.end()) ∨
    isspace(cp[1]) ∨ ((cp[1] & #FF) ≡ C_RIGHT_POINTING_DOUBLE_ANGLE_QUOTATION_MARK) ∨ (cp[1] ≡
    ', '))) {
    o += "{\\small~}";
    o += c;
}
else { /* Characters requiring protection against interpretation as control sequences. */
    if (quotedCharacters.find_first_of(c) ≠ string::npos) {
        o += '\\';
    }
    if (¬inmath ∧ (quotedTextCharacters.find_first_of(c) ≠ string::npos)) {
        o += '\\';
    }
    o += c;
}
```

This code is used in section 77.

90. Special commands which define text substitutions are recognised by this method, parsed, and added to the substitution list.

⟨ Class definitions 8 ⟩ +≡

```

bool LaTeXGenerationFilter::isSubstitution(string cmd)
{
    string s = etextBodyParserFilter::specialCommand(cmd);
    bool isSub = false, bogus = true;
    unsigned int n, m, l;
    char delim;
    if (s.find("Substitute_") == 0) {
        isSub = true;
        s = s.substr(11);
        n = s.find_first_not_of(' ');
        if (n != string::npos) {
            delim = s[n];
            m = s.find(delim, n + 1);
            if (m != string::npos) {
                l = s.find(delim, m + 1);
                if (l != string::npos) {
                    bogus = false;
                    transformer.addSubstitution(s.substr(n + 1, (m - n) - 1), s.substr(m + 1, (l - m) - 1));
                }
            }
        }
    }
    if (bogus) {
        issueMessage("Invalid_LaTeX_Substitute_special", cerr);
        issueMessage(auditFilter::quoteArbitraryString(cmd), cerr);
    }
}
return isSub;
}

```

91. HTML Generation.

This sink translates parsed body copy (emitted by *etextBodyParser*) into HTML. An index document is written, which contains links to the individual chapters which, in turn, link to one another and an additional document containing any footnotes. When the sink is created, it must be passed the *basename* used to generate the names for these files; the *basename* may not be “-” denoting standard output.

HTML generation is a sink (or, more precisely, a variable-tentacled squid with built-in sinks for each) because, unless instantiated with the *singleFile* option, it generates an index file, a file for each chapter, and a footnote file if needed. Since there’s no way to know which files will be needed prior to processing the text, higher level code can’t be expected to create a forked pipeline to write all these files. So, *HTMLGenerationSink* simply consumes the parsed document body it receives and creates whatever files are required to build an HTML document from the text on the fly.

⟨Class definitions 8⟩ +≡

```
class HTMLGenerationSink : public textSink {
private:
    static const int defaultFootnotePad = 60;    /* Default blank lines between footnotes */
    bool italics, inmath, infoot, firstchap, hastitle, hasauthor;
    string basename;    /* Document base name */
    string stime;    /* Processing date and time for comment */
    string indexFileName;    /* Index (or main document if singleFile name */
    ofstream *index;    /* Index output stream */
    ofstream *chap;    /* Chapter output stream */
    ofstream *foot;    /* Footnote output stream */
    string htitle;    /* HTML <title> of document */
    string hauthor;    /* Author of document */
    int chapno;    /* Chapter index (1...n) */
    string chapnumber;    /* Chapter number from text */
    string chapname;    /* Chapter name */
    string qchapnumber, qchapname;    /* Chapter number and name in queue */
    string chtitle;    /* Generated HTML title for chapter document */
    bool singleFile;    /* Make one HTML file for entire document */
    string footdocname;    /* Footnote document name */
    int footnum;    /* Footnote number */
    int footnest;    /* Footnote nesting level */
    bool fitalics;    /* Save italic mode while processing footnote */
    bool breakPending;    /* Float-clearing break pending ? */
    int footnotePad;    /* Footnote pad lines */
    queue<string> chapterCache;    /* Pending chapter cache */
    int indexline, chapline, footline;    /* Index, chapter, and footnote line counters */
    deque<string> declarationsQueue;    /* HTML special declarations */
    string quoteHTMLString(string s);
    string translateHTMLString(string t);
    void emitq(string s)
    {
        emit(translateHTMLString(s));
    }
    void generateAlignedParagraph(string alignment, char bracket, string text, string
        terminator = "<br>");
    void writeHTMLDocumentPreamble(ostream &os, string title, int *lineCounter);
    void writeHTMLDocumentBodyStart(ostream &os, string title, int *lineCounter);
    void writeHTMLDocumentPostamble(ostream &os, int *lineCounter);
    static void createNavButton(string button, const unsigned char source[], int length);
```

```

static void createNavButtons(void);
void createNavigationPanel(int prev, int next, bool inParagraph = true);
virtual void emit(string s, textComponent *destination =  $\Lambda$ )
{
    if (singleFile) {
        *index << s << "\n";
        indexline++;
    }
    else {
        if (infoot  $\vee$  (chap  $\equiv \Lambda$ )) {
            *(infoot ? foot : index) << s << "\n";
            if (infoot) {
                footline++;
            }
            else {
                indexline++;
            }
        }
        else {
            chapterCache.push(s + "\n");
        }
    }
    lineNumber++;
}

static string pruneIndent(string s)
{
    assert(s  $\neq$  "");
    return s.substr(s.find_first_not_of(' '),);
}

static string elideNewLines(string s)
{
    string o = s;
    unsigned int i;
    while ((i = o.find('\n'))  $\neq$  string::npos) {
        o.replace(i, 1, "");
    }
    while ((o.length() > 0)  $\wedge$  (o[o.length() - 1]  $\equiv$  ' ')) {
        o = o.substr(0, o.length() - 1);
    }
    return o;
}

static unsigned int linesIn(string s)
{
    /* Count lines in string */
    return count(s.begin(), s.end(), '\n');
}

void flushBreak(void)
{
    if (breakPending) {
        emit("<br_clear=all>");
        breakPending = false;
    }
}

```

```

    }
public:
    void setFootnotePad(int fp = defaultFootnotePad)
    {
        footnotePad = fp;
    }
    int getFootnotePad(void)
    {
        return footnotePad;
    }
    HTMLGenerationSink(string bname, bool make_one_file = false)
    {
        time_t t = time( $\Lambda$ );
        stime = ctime(&t);
        stime = stime.substr(0, stime.length() - 1);
        italics = inmath = infoot = false;
        hastitle = hasauthor = false;
        firstchap = false;
        if (bname == "-") {
            cerr << "Cannot write HTML document set to standard output.\n";
            exit(1);
        }
        basename = bname;
        singleFile = make_one_file;
        index = chap = foot =  $\Lambda$ ;
        footnum = footnest = 0;
        footdocname = "";
        setFootnotePad();
        breakPending = false;
        chapno = 0;
        indexline = chapline = footline = 0;
    }
    virtual ~HTMLGenerationSink()
    {}
    string componentName(void)
    {
        return "HTMLGenerationSink";
    }
    string getBaseName(void)
    {
        return basename;
    }
    void put(string s);
};

```

92. The *put* method of the `HTMLGenerationSink` wraps HTML tags around the line-level structure of the text to achieve the desired formatting. Since almost all of the real work is done upstream (by `etextBodyParserFilter`) and downstream (by *translateHTMLString*) there is actually little that needs doing here.

⟨ Class definitions 8 ⟩ +=

```
void HTMLGenerationSink::put(string s)
{
    bodyState state = DecodeBodyState(s[0]);
    char bracket = s[1];
    string text = s.substr(2);
    ostreamstream efn;
    switch (state) {
    case BeginText: indexFileName = basename + ".html";
        index = new ofstream(indexFileName.c_str(), ios::out);
        break;
    case Declarations: ⟨ Process declarations in HTML 93 ⟩;
    case DocumentTitle: ⟨ Process document title in HTML 94 ⟩;
    case Author: ⟨ Process author in HTML 95 ⟩;
    case ChapterNumber: ⟨ Process chapter number in HTML 97 ⟩;
    case ChapterName: ⟨ Process chapter name in HTML 98 ⟩;
    case InTextParagraph: generateAlignedParagraph("justify", bracket, text, "");
        break;
    case InBlockQuote: generateAlignedParagraph("quote", bracket, text, "");
        break;
    case InRaggedRight: generateAlignedParagraph("left", bracket, text);
        break;
    case InRaggedLeft: generateAlignedParagraph("right", bracket, text);
        break;
    case InPreformattedTable: generateAlignedParagraph("table", bracket, text, "");
        break;
    case InCentred: generateAlignedParagraph("center", bracket, text);
        break;
    case EndOfText:
        if (¬singleFile) {
            *index << "</table>\n";
            indexline++;
        }
        *index << "</div>\n";
        indexline++;
        writeHTMLDocumentPostamble(*index, &indexline);
        index->close();
        ⟨ Complete chapter file generation in HTML 101 ⟩;
    if (foot ≠ Λ) {
        *foot << "</div>\n";
        footline++;
        writeHTMLDocumentPostamble(*foot, &footline);
        foot->close();
        if (verbose) {
            cerr << footdocname << ":␣" << footline << "␣lines.\n";
        }
    }
    }
    if (verbose) {
```

```

    cerr << indexFileName << ":" << indexline << "_lines.\n";
}
break;
default: cerr << "***_State_" << stateNames[state] << "_" << bracket <<
    "_not_handled_in_HTMLGenerationSink***\n";
    exit(1);
}
}

```

93. HTML-specific declarations are saved in *declarationsQueue* whence they are emitted in the `<head>` section of each HTML file generated.

⟨Process declarations in HTML 93⟩ ≡

```

if (bracket ≡ Body) {
    assert(etextBodyParserFilter::isLineSpecial(text));
    declarationsQueue.push_back(etextBodyParserFilter::specialCommand(text));
}
break;

```

This code is used in section 92.

94. When we see the title, we save it in *htitle* for use in the `<title>` tag of each of the HTML documents we generate. Titles may span multiple lines; we concatenate them into a single line in *htitle*.

⟨Process document title in HTML 94⟩ ≡

```

switch (bracket) {
case Begin: htitle = "";
    break;
case Body:
    if (htitle ≠ "") {
        htitle += "_";
    }
    htitle += quoteHTMLString(pruneIndent(text));
    hastitle = true;
    break;
case Void: hastitle = false;
    htitle = "";
    break;
}
break;

```

This code is used in section 92.

95. Once we've seen the author or received the *Void* notification that no author was given, we're ready to generate the header for the *index* document. The author specification may also span multiple lines, which are concatenated into *hauthor*.

```

⟨Process author in HTML 95⟩ ≡
  switch (bracket) {
  case Begin: hauthor = "";
    break;
  case Body:
    if (hauthor ≠ "") {
      hauthor += "␣";
    }
    hauthor += quoteHTMLString(pruneIndent(text));
    break;
  case End: hasauthor = true;
    /* Note fall-through */
  case Void: ⟨Generate index document header in HTML 96⟩;
    break;
  }
  break;

```

This code is used in section 92.

96. The HTML index document is a list of links to the individual chapter documents. Generate the canned HTML header, the title and author information (if any), and begin the list of chapters.

```

⟨Generate index document header in HTML 96⟩ ≡
  writeHTMLDocumentPreamble(*index, htitle, &indexline);
  writeHTMLDocumentBodyStart(*index, htitle, &indexline);
  *index << "<div␣class=\"bodycopy\">\n";
  indexline++;
  if (hastitle) {
    *index << "<h1␣align=center>" << translateHTMLString(htitle) << "</h1>\n";
    indexline++;
  }
  if (hasauthor) {
    *index << "<h2␣align=center>" << translateHTMLString(hauthor) << "</h2>\n";
    indexline++;
  }

```

This code is used in section 95.

97. We save the chapter number for later use in the index document or chapter titles written in *singleFile* mode.

⟨Process chapter number in HTML 97⟩ ≡

```
switch (bracket) {
  case Begin: chapnumber = "";
    break;
  case Body: chapnumber += quoteHTMLString(pruneIndent(text)) + "\n";
    break;
  case Void: chapnumber = "";
    /* Note fall-through */
  case End: break;
}
break;
```

This code is used in section 92.

98. Unless *singleFile* is set, each chapter is written into its own HTML document named *basename_chapn.html*, linked to the *index* document. In *singleFile* we simply generate a chapter break within the unified document file.

⟨Process chapter name in HTML 98⟩ ≡

```
switch (bracket) {
  case Begin: chapname = "";
    break;
  case Body: chapname += quoteHTMLString(pruneIndent(text)) + "\n";
    break;
  case Void: chapname = "";
    /* Note fall-through */
  case End: flushBreak();
    if (singleFile) {
      ⟨Generate chapter title for single file output in HTML 99⟩;
    }
    else {
      ⟨Generate chapter title for document tree output in HTML 100⟩;
    }
    break;
}
break;
```

This code is used in section 92.

99. When a single HTML document is being generated containing all chapters, the chapter break simply causes the generation of a title within the output document. The title contains the chapter number and chapter name given in the chapter break sequence with, if both are given, a horizontal rule separating them. If a chapter break specifies neither a chapter number nor title, only the horizontal rule is generated, while if only a number or title appear, no rule is output.

⟨Generate chapter title for single file output in HTML 99⟩ ≡

```

*index << "<h2 align=center style=\"margin-left: 5%; margin-right: 5%;\">\n";
indexline++;
if (chapnumber ≠ "") {
  *index << translateHTMLString(chapnumber);
  indexline += linesIn(chapnumber);
}
if (((chapname ≠ "") ∨ (chapname ≠ "")) ∨ ((chapname ≡ "") ∧ (chapname ≡ ""))) {
  *index << "<hr width=\"25%\" size=2 noshade>\n";
  indexline++;
}
if (chapname ≠ "") {
  *index << translateHTMLString(chapname);
  indexline += linesIn(chapname);
}
*index << "</h2>\n";
indexline++;

```

This code is used in section 98.

100. When generating a multiple-file HTML document tree from an Etext, one chapter per file, a chapter break is understandably more of an event. We need to close out the last chapter (if any), open a new HTML document for the new chapter, add an index entry pointing to it in the *index* document, and crank out the HTML preamble boilerplate and chapter title for the next chapter. If this is the first chapter mark, we create the GIF files used as the navigation buttons in the chapter documents.

```
<Generate chapter title for document tree output in HTML 100> ≡
<Complete chapter file generation in HTML 101>;
if (chapno ≡ 0) {
    createNavButtons(); /* Create GIF navigation buttons */
    *index << "<table width=\"80%\" cols=3 align=center>\n"; /* Begin chapter table */
    indexline++;
}
chapno++;
chapline = 0;
efn << basename << "_chap" << chapno << ".html";
chap = new ofstream(efn.str().c_str(), ios::out);
chtitle = htitle;
if (chapnumber ≠ "") {
    chtitle += ":\<";
    chtitle += chapnumber;
}
qchapnumber = chapnumber;
qchapname = chapname; /* Create link to chapter in index document */
*index << "<tr><th align=right width=\"15%\"><a href=\"" << efn.str() << "\>";
if (chapnumber ≡ "") {
    *index << chapno << ".";
}
else {
    *index << elideNewLines(chapnumber);
    indexline += linesIn(elideNewLines(chapnumber));
}
*index << "</a><td width=\"5%\">&nbsp;<td width=\"80%\"><a href=\"" << efn.str() <<
    "\>" << translateHTMLString(elideNewLines(chapname)) << "</a>\n";
indexline += linesIn(elideNewLines(chapname)) + 1;
```

This code is used in section 98.

101. Complete generation of the current chapter document and close the file. This may be called even if no chapter document is open, for example, when a single file document is being written. `<link>` tags are included in the header to indicate the order of the chapters and their relationship to the parent index document.

(Complete chapter file generation in HTML 101) \equiv

```

if (chap  $\neq$   $\Lambda$ ) {
  string s;
  writeHTMLDocumentPreamble(*chap, chtitle, &chapline);
  *chap  $\ll$  "<link_href=\"\"  $\ll$  indexFileName  $\ll$  "\"_rel=parent_rev=child>\n";
  chapline ++;
  if (chapno > 1) {
    *chap  $\ll$  "<link_href=\"\"  $\ll$  basename  $\ll$  "_chap"  $\ll$  (chapno - 1)  $\ll$  ".html"  $\ll$ 
      "\"_rel=prev_rev=next>\n";
    chapline ++;
  }
  if (state  $\neq$  EndOfText) {
    *chap  $\ll$  "<link_href=\"\"  $\ll$  basename  $\ll$  "_chap"  $\ll$  (chapno + 1)  $\ll$  ".html"  $\ll$ 
      "\"_rel=next_rev=prev>\n";
    chapline ++;
  }
  writeHTMLDocumentBodyStart(*chap, chtitle, &chapline);
  if (hastitle) {
    *chap  $\ll$  "<table_width=\"100%\">\n";
    *chap  $\ll$  "<tr><td_width=\"25%\"_ valign=top>&nbsp;\n";
    *chap  $\ll$  "<td_width=\"50%\"_ align=center><h1>"  $\ll$  htitle  $\ll$  "</h1>\n";
    *chap  $\ll$  "<td_width=\"25%\"_ align=right>\n";
    createNavigationPanel(chapno - 1, (state  $\equiv$  EndOfText) ? 0 : (chapno + 1), false);
    *chap  $\ll$  "</table>\n";
    chapline += 5;
  }
  else {
    createNavigationPanel(chapno - 1, (state  $\equiv$  EndOfText) ? 0 : (chapno + 1));
  }
  *chap  $\ll$  "<div_class=\"bodycopy\">\n";
  *chap  $\ll$  "<h1_align=center_style=\"margin-left:5%;_margin-right:5%;\">\n";
  chapline += 2;
  if (qchapnumber  $\neq$  "") {
    *chap  $\ll$  translateHTMLString(qchapnumber);
    chapline += linesIn(qchapnumber);
  }
  if (((qchapname  $\neq$  "")  $\vee$  (qchapname  $\neq$  ""))  $\vee$  ((qchapname  $\equiv$  "")  $\wedge$  (qchapname  $\equiv$  ""))) {
    *chap  $\ll$  "<hr_width=\"25%\"_ size=2_noshade>\n";
    chapline ++;
  }
  if (qchapname  $\neq$  "") {
    *chap  $\ll$  translateHTMLString(qchapname);
    chapline += linesIn(qchapname);
  }
  *chap  $\ll$  "</h1>\n";
  *chap  $\ll$  "\n";
  chapline += 2;
  while ( $\neg$ chapterCache.empty()) {

```

```

    *chap << chapterCache.front();
    chapline++;
    chapterCache.pop();
}
*chap << "</div>\n";
chapline += 2;
createNavigationPanel(chapno - 1, (state ≡ EndOfText) ? 0 : (chapno + 1));
writeHTMLDocumentPostamble(*chap, &chapline);
chap->close();
if (verbose) {
    cerr << basename << "_chap" << chapno << ".html:_" << chapline << "_lines.\n";
}
}

```

This code is used in sections [92](#) and [100](#).

102. This function handles the various kinds of aligned paragraphs we encounter in a document. It precedes the body of the paragraph with a `<p>` tag with the specified alignment. Block quotes are also handled here using the pseudo-alignment of “quote”—they are wrapped by a `<blockquote>` tag instead.

(Class definitions 8) +=

```
void HTMLGenerationSink::generateAlignedParagraph(string alignment, char bracket, string
    text, string terminator = "<br>")
{
    string s;
    switch (bracket) {
    case Begin: emit(""); /* Purely for readability when hand-editing HTML */
        if (alignment == "quote") {
            emit("<blockquote>");
        }
        else if (alignment == "table") {
            emit("<pre>");
        }
        else {
            emit("<p align=\"\" + alignment + \"\">");
        }
        break;
    case Body:
        if (etextBodyParserFilter::isLineSpecial(text)) {
            emit(etextBodyParserFilter::specialCommand(text));
        }
        else {
            s = translateHTMLString(text);
            if (!infoot) {
                s += terminator;
            }
            emit(s);
        }
        break;
    case End:
        if (alignment == "quote") {
            emit("</blockquote>");
        }
        else if (alignment == "table") {
            emit("</pre>");
        }
        else {
            emit("</p>");
        }
        break;
    case Void: break;
    }
}
```

103. Quote string *s* for output to HTML, replacing HTML metacharacters with their corresponding “&” entities. With typical text it would probably be faster to first test for the presence of any of the metacharacters in a line using *find_first_of*, returning the line unmodified if none were found. This would only require character-by-character examination in the normally rare circumstance where the line contains a character we need to quote. Given that this program is extremely unlikely to be run frequently, we’ll forego such refinements in the interest of clarity.

⟨ Class definitions 8 ⟩ +≡

```
string HTMLGenerationSink::quoteHTMLString(string s)
{
    string::iterator cp;
    string o = "";
    for (cp = s.begin(); cp < s.end(); cp++) {
        int c = (*cp) & #FF;
        switch (c) {
            case '&': o += "&amp;";
                break;
            case '<': o += "&lt;";
                break;
            case '>': o += "&gt;";
                break;
            default:
                if (c < ' ') {
                    ⟨ Quote control character in HTML 104 ⟩;
                }
                else {
                    o += c;
                }
                break;
        }
    }
    return o;
}
```

104. This is a control character. Emit as *^letter* unless it is considered as white space (for example, carriage return and line feed), in which case it’s sent directly to the output.

⟨ Quote control character in HTML 104 ⟩ ≡

```
if (isspace(c)) {
    o += c;
}
else {
    o += "^" + ('@' + c);
}
```

This code is used in section 103.

105. Translate the text string argument into HTML, processing control sequences as required. Note that metacharacters are expanded by the call on *quoteHTMLString* right at the top of the function, so the balance of the code needn't worry about quoting them. That leaves italic and math mode toggles, and footnotes to be handled here.

```

⟨Class definitions 8⟩ +=
string HTMLGenerationSink::translateHTMLString(string t)
{
    string::iterator cp;
    string o = "";
    char c;
    static const string punctuation = PUNCTUATION;
    string s = quoteHTMLString(t);
    for (cp = s.begin(); cp < s.end(); cp++) {
        c = *cp;
        if ( $\neg$ inmath  $\wedge$  c  $\equiv$  '_' ) {
            ⟨Toggle italic text mode in HTML 106⟩;
        }
        else if (c  $\equiv$  '\\'  $\wedge$  ((cp + 1) < s.end())  $\wedge$  ((cp[1]  $\equiv$  '(')  $\vee$  (cp[1]  $\equiv$  ')')) {
            ⟨Toggle math mode in HTML 107⟩;
        }
        else if ( $\neg$ inmath  $\wedge$  ((cp + 2) < s.end())  $\wedge$  ((c  $\equiv$  '[')  $\vee$  ((c  $\equiv$  '␣')  $\wedge$  (cp[1]  $\equiv$  '['))  $\vee$  ((c  $\equiv$  '␣')  $\wedge$  (cp[1]  $\equiv$  '␣')  $\wedge$  (cp[2]  $\equiv$  '['))))) {
            ⟨Begin footnote in HTML 108⟩;
        }
        else if ( $\neg$ inmath  $\wedge$  c  $\equiv$  ']') {
            ⟨End footnote in HTML 109⟩;
        }
        else {
            ⟨Output text character in HTML 111⟩;
        }
    }
    return o;
}

```

106. The underscore character, “_”, toggles text between the normal roman and italic fonts.

```

⟨Toggle italic text mode in HTML 106⟩  $\equiv$ 
    italics =  $\neg$ italics;
    if (italics) {
        o += "<i>␣";
    }
    else {
        o += "</i>";
    }

```

This code is used in section 105.

107. HTML doesn't support mathematics, at least not without plug-ins or XML horrors few users at this writing are likely to have installed. When we encounter mathematics in the text, we simply enclose it in a `<table>` box, tinted pink to indicate it requires attention, and proceed. The editor of the document can then use `TeXToGIF` or an equivalent tool to render the equation for Web publication.

```

<Toggle math mode in HTML 107> ≡
  inmath = cp[1] ≡ '(';
  ° cp++;
  if (inmath) {
    o += "<table_bgcolor=\"#FFA0A0\"><tr><td>";
  }
  else {
    o += "</table>";
  }

```

This code is used in section 105.

108. Footnotes appear in-line, within [square brackets]. If we're writing a single file, we simply output them in-line, in square brackets, using a small sans-serif font with a yellow background (assuming the browser comprehends such things, naturally). When generating a multiple file document tree, footnotes are placed in a dedicated file, with a link to that file and the fragment ID of the footnote in the body copy where the note appeared. The footnote link is the footnote number as a superscript. We don't support nested footnotes. If the input document contains them, we render nested footnotes in-line in the same manner as outer level footnotes in *singleFile* mode.

Browsers which support targeted windows will open the footnote document in a new window which will be scrolled as subsequent footnote links are clicked.

⟨Begin footnote in HTML 108⟩ ≡

```
#define NETSCRAPE_SUCKS /* Work around moronic style/table interaction in Netscap */
if (footnest > 0) {
    issueMessage("Cannot_nest_footnotes_in_HTML_document_output.");
    o += "<span_style=\"background-color: #FFFA0\">[<small>";
    o += "<font_face=\"Helvetica, Arial\">";
}
else {
    if (singleFile) {
#ifdef NETSCRAPE_SUCKS
        o += "<span_style=\"background-color: #FFFA0\">[<small>";
        o += "<font_face=\"Helvetica, Arial\">";
#else
        flushBreak();
        o += "<sup>*</sup>\n<table_width=\"25%\"_align=right_hspace=6_bgc\
            olor=\"#FFFFD0\">\n";
        o += "<tr><td><b>*</b>_<small>\n";
#endif
    }
    else {
        ostringstream eflink;
        ⟨Create footnote file for first footnote in HTML 110⟩;
        footnum++;
        eflink << "<a_href=\"\" << footdocname << "#" << footnum << "\"_target=\"\" << basename <<
            "_foot\">\" << "<sup>\" << footnum << "</sup></a>";
        o += eflink.str();
        emit(o);
        o = "";
        infoot = true;
        eflink.str("");
        eflink << "<a_name=\"\" << footnum << "\">\" << footnum << ".</a>_";
        o += eflink.str();
    }
    fitalics = italics;
    italics = false;
}
if ((c ≡ ' ') ^ ((cp + 1) < s.end())) {
    if (cp[1] ≡ ' ') {
        cp++;
    }
    cp++;
}
footnest++;
```

This code is used in section 105.

109. Close a footnote when the right bracket is encountered. When generating a separate footnote document, we use a `<pre>` element with *footnotePad* blank lines following the footnote so the next one won't appear in the window for typical browser window sizes.

⟨End footnote in HTML 109⟩ ≡

```

    if (footnest ≡ 1) {
        if (singleFile) {
#ifndef NETSCRAPE_SUCKS
            o += "</font></small>]</span>";
#else
            o += "\n</small>\n</table>\n";
#endif
            breakPending = true;
        }
        else {
            int l;
            emit(o);
            emit("<p>");
            emit("<pre>");
            for (l = 0; l < footnotePad; l++) {
                emit("");
            }
            emit("</pre>");
            o = "";
            infoot = false;
        }
        italics = fitalics;
    }
    else if (footnest > 1) {
        o += "</font></small>]</span>";
    }
    if (footnest ≡ 0) {
        issueMessage("Mismatched_end_of_footnote_\["\]\_bracket.");
    }
    else {
        footnest--;
    }

```

This code is used in section 105.

110. Upon encountering a footnote while creating a multiple file HTML document, we check whether a footnote document has already been created. If not, this is the first footnote; a footnote document is created to receive it and any subsequent footnotes.

⟨ Create footnote file for first footnote in HTML 110 ⟩ ≡

```

if (foot ≡  $\Lambda$ ) {
  footdocname = basename + "_foot" + ".html";
  foot = new ofstream(footdocname.c_str(), ios::out);
  writeHTMLDocumentPreamble(*foot, htitle + ":\Notes", &footline);
  writeHTMLDocumentBodyStart(*foot, htitle + ":\Notes", &footline);
  *foot << "<div\class=\"bodycopy\">\n";
  footline++;
}

```

This code is used in section 108.

111. Output a text character. Since we've already handled quoting of metacharacters, the only thing we need to worry about here is adding a nonbreaking space around eligible punctuation if *frenchPunct* is set.

⟨ Output text character in HTML 111 ⟩ ≡

```

if ( $\neg inmath \wedge frenchPunct \wedge ((c \& \#FF) \equiv C\_LEFT\_POINTING\_DOUBLE\_ANGLE\_QUOTATION\_MARK) \wedge (cp \neq$ 
  s.end())  $\wedge (\neg isspace(cp[1]))$ ) {
  o += c;
  o += "&nbsp;";
}
else if ( $\neg inmath \wedge frenchPunct \wedge (punctuation.find\_first\_of(c) \neq$ 
  string::npos)  $\wedge (((cp + 1) \equiv s.end()) \vee isspace(cp[1]) \vee ((cp[1] \& \#FF) \equiv$ 
   $C\_RIGHT\_POINTING\_DOUBLE\_ANGLE\_QUOTATION\_MARK) \vee (cp[1] \equiv ', ')))$ ) {
  o += "&nbsp;";
  o += c;
}
else {
  o += c;
}

```

This code is used in section 105.

112. HTML documents have a stereotyped preamble. This function writes the preamble at the start of a document we're writing to a stream.

(Class definitions 8) +=

```
void HTMLGenerationSink::writeHTMLDocumentPreamble(ostream &os, string title, int
    *lineCounter)
{
    const int preambleLines = 11;
    deque<string>::iterator decl;
    os << "<!DOCTYPE_HTML_PUBLIC_\"-//W3C//DTD_HTML_3.2_Final//EN\">\n";
    os << "<html_version=\"-//W3C//DTD_HTML_3.2_Final//EN\">\n";
    os << "<!--_Translated_by_\" PRODUCT \"_\" VERSION \"_\" REVDATE \"_)_on_\" + stime + \"-->\n";
    os << "<head>\n";
    os << "<title>" << title << "</title>\n";
    if (hastitle) {
        os << "<meta_name=\"description\"_content=\"\" << elideNewLines(title) << "\">\n";
        *lineCounter += 1;
    }
    if (hasauthor) {
        os << "<meta_name=\"author\"_content=\"\" << elideNewLines(hauthor) << "\">\n";
        *lineCounter += 1;
    }
    os << "<style_type=\"text/css\">\n";
    os << "DIV.bodycopy{\n";
    os << "margin-left:15%;\n";
    os << "margin-right:10%;\n";
    os << "}\n";
    os << "</style>\n";
    *lineCounter += preambleLines;
    for (decl = declarationsQueue.begin(); decl != declarationsQueue.end(); decl++) {
        os << *decl << "\n";
        *lineCounter += 1;
    }
}
```

113. After the HTML preamble comes the sequence which ends the header and begins the body of the document. We write this in a separate function to allow declarations and links to be added before the end of the header.

(Class definitions 8) +=

```
void HTMLGenerationSink::writeHTMLDocumentBodyStart(ostream &os, string title, int
    *lineCounter)
{
    const int bodyStartLines = 3;
    os << "</head>\n";
    os << "\n";
    os << "<body_bgcolor=\"#FFFFFF\">\n";
    *lineCounter += bodyStartLines;
}
```

114. HTML documents similarly close with a stereotyped postamble. This function appends the boilerplate to the end of a stream.

(Class definitions 8) +=

```
void HTMLGenerationSink::writeHTMLDocumentPostamble(ostream &os, int *lineCounter)
{
    const int postambleLines = 2;
    os << "</body>\n";
    os << "</html>\n";
    *lineCounter += postambleLines;
}
```

115. Each HTML chapter document contains a navigation panel consisting of “next”, “previous”, and “up” buttons, the first two of which are replaced by greyed-out versions for the last and first chapters respectively. These reference the GIF image buttons created by *createNavButtons*.

createNavigationPanel writes a navigation panel to the current chapter document, assumed open as output stream *chap*. The numbers of the previous and next chapters are given by the *prev* and *next* arguments, which are zero if no such chapter exists—in that case a disabled (greyed-out) button with no link appears in the panel.

(Class definitions 8) +=

```
void HTMLGenerationSink::createNavigationPanel(int prev, int next, bool inParagraph)
{
    /* Previous chapter button. */
    if (inParagraph) {
        *chap << "<p align=right>\n";
    }
    if (prev != 0) {
        *chap << "<a href=\"\" << basename << "_chap" << prev << ".html\">" <<
            "<img align=middle src=\"prev.gif\" \"height=32 width=32 border=0 alt=\"Previous\"></a> \n";
    }
    else {
        *chap << "<img align=middle src=\"prev_gr.gif\" \"height=32 width=32 \n"
            "border=0 alt=\"\"> \n";
    }
    /* Up to table of contents button. */
    *chap << "<a href=\"\" << basename << ".html\">" << "<img align=middle src=\"up.gif\" \n"
        "\"height=32 width=32 border=0 alt=\"Contents\"></a> \n";
    /* Next chapter button. */
    if (next != 0) {
        *chap << "<a href=\"\" << basename << "_chap" << next << ".html\">" << "<img align=middle \n"
            "src=\"next.gif\" \"height=32 width=32 border=0 alt=\"Next\"></a>\n";
    }
    else {
        *chap << "<img align=middle src=\"next_gr.gif\" \"height=32 width=32 \n"
            "border=0 alt=\"\">\n";
    }
    if (inParagraph) {
        *chap << "</p>\n";
    }
    chapline += 3 + (inParagraph ? 2 : 0);
}
```

116. This **static** function creates the GIF files for the buttons in the navigation panel. It is called when we are creating a multi-file document tree and encounter the first chapter title.

```
<Class definitions 8> +=
void HTMLGenerationSink::createNavButtons(void)
{
    <Definition of navigation buttons in HTML 118>;
#ifdef FOOTNOTE_BUTTON_NEEDED
    createNavButton("foot", d_foot, sizeof d_foot);    /* Footnote */
#endif
    createNavButton("next", d_next, sizeof d_next);    /* Next */
    createNavButton("prev", d_prev, sizeof d_prev);    /* Previous */
    createNavButton("up", d_up, sizeof d_up);          /* Up (to Table of Contents) */
    createNavButton("next_gr", d_next_gr, sizeof d_next_gr);
        /* Greyed out Next (for last chapter) */
    createNavButton("prev_gr", d_prev_gr, sizeof d_prev_gr);
        /* Greyed out Prev (for first chapter) */
}
```

117. The GIF navigation buttons in the HTML chapter documents are created by calling *createNavButton* for each button definition embedded by the **#include** in the following section. *createNavButtons* calls this function for each button file. There's actually nothing at all GIF-specific about this function—it just writes out arbitrary binary data from memory to an **ofstream**.

```
<Class definitions 8> +=
void HTMLGenerationSink::createNavButton(string button, const unsigned char source[], int
    length)
{
    string buttonFile;
    ofstream *bf;
    buttonFile = button + ".gif";
    bf = new ofstream(buttonFile.c_str(), ios::out | ios::binary);
    bf->write(source, length);
    bf->close();
}
```

118. When generating HTML output, we want to include language-neutral navigation buttons. These are small GIF images, which present a problem if we wish to distribute this program in text form. To avoid the need for system-specific text to binary and/or archive extraction utilities, we simply embed the GIF images in this file as binary data definitions and write them into the HTML document directory.

```
<Definition of navigation buttons in HTML 118> ≡
#include "buttons.h"
```

This code is used in section 116.

119. Palm Markup Language Generation.

This filter translates parsed body copy (emitted by *etextBodyParser*) into Palm Markup Language source code, which it passes down the pipeline.

⟨Class definitions 8⟩ +≡

```

class PalmGenerationFilter : public textFilter {
private:
    bool italics, inmath, quoth, hastitle, hasauthor, infoot, intable, firstchap;
    string htitle;    /* Title of document */
    string hauthor;    /* Author of document */
    string chapnumber; /* Chapter number from text */
    string chapname;   /* Chapter name */
    string partext;    /* Paragraph accumulation string */
    int parline;       /* Paragraph line counter */
    int chapno;        /* Chapter number (for anonymous chapters) */
    int footnum;       /* Footnote number */
    int footnest;      /* Footnote nesting level */
    string footnotes;  /* Footnotes saved for output at end */
    string footpar;    /* Footnote paragraph accumulator */
    string footsave;   /* Save paragraph during footnote accumulation */
    bool fitalics, fquoth; /* Text processing modes saved during footnote */
    textSubstituter transformer; /* Text substituter for substitute specials */
    string quotePalmString(string s);
    static string pruneIndent(string s)
    {
        assert(s ≠ "");
        return s.substr(s.find_first_not_of(' '),);
    }
    virtual void emit(string s, textComponent *destination = Λ)
    {
        if (infoot) {
            footnotes += s + "\n";
        }
        else {
            textFilter::emit(s, destination);
        }
    }
    void emitq(string s)
    {
        emit(quotePalmString(s));
    }
    void generateFilledParagraph(bodyState state, string envtype, char bracket, string text);
    void generateAlignedParagraph(bodyState state, string envtype, char bracket, string text);
    bool isSubstitution(string cmd);
public:
    PalmGenerationFilter()
    {
        italics = inmath = quoth = false;
        hastitle = hasauthor = false;
        intable = firstchap = infoot = false;
        footnest = footnum = 0;
        chapno = 0;
    }

```

```
    }  
    virtual ~PalmGenerationFilter()  
    {}  
    string componentName(void)  
    {  
        return "PalmGenerationFilter";  
    }  
    void put(string s);  
};
```

120. The *put* method of the *PalmGenerationFilter* wraps Palm Markup Language commands around the line-level structure of the text to achieve the desired formatting. Since almost all of the real work is done upstream (by *etextBodyParserFilter*) and downstream (by *quotePalmString*) there is relatively little that needs doing here.

⟨ Class definitions 8 ⟩ +≡

```

void PalmGenerationFilter::put(string s)
{
    bodyState state = DecodeBodyState(s[0]);
    char bracket = s[1];
    string text = s.substr(2);
    switch (state) {
    case BeginText: ⟨ Generate start of document in Palm 121 ⟩;
    case Declarations: ⟨ Process declarations in Palm 122 ⟩;
    case DocumentTitle: ⟨ Process document title in Palm 123 ⟩;
    case Author: ⟨ Process author in Palm 124 ⟩;
    case ChapterNumber: ⟨ Process chapter number in Palm 125 ⟩;
    case ChapterName: ⟨ Process chapter name in Palm 126 ⟩;
    case InTextParagraph: generateFilledParagraph(state, "", bracket, text);
        break;
    case InBlockQuote: generateFilledParagraph(state, "\\t", bracket, text);
        break;
    case InRaggedRight: generateAlignedParagraph(state, "", bracket, text);
        break;
    case InRaggedLeft: generateAlignedParagraph(state, "\\r", bracket, text);
        break;
    case InPreformattedTable:
        if (bracket ≡ Begin) {
            intable = true;
        }
        generateAlignedParagraph(state, "", bracket, text);
        if (bracket ≡ End) {
            intable = false;
        }
        break;
    case InCentred: generateAlignedParagraph(state, "\\c", bracket, text);
        break;
    case EndOfText:
        if (footnum > 0) {
            emit(footnotes);    /* Append footnotes to document */
        }
        if (verbose) {
            cerr << "Palm:_" << (getLineNumber() + count(footnotes.begin(), footnotes.end()),
                '\n') << "_lines_output.\n";
        }
        break;
    default: cerr << "***_State_" << stateNames[state] << "_" << bracket <<
        "_not_handled_in_PalmGenerationFilter_***\n";
        exit(1);
    }
}

```

121. Generate the boilerplate at the start of a Palm Markup Language document.

⟨Generate start of document in Palm 121⟩ ≡

```
{
  time_t t = time( $\Lambda$ );
  string stime = ctime(&t);
  stime = stime.substr(0, stime.length() - 1);
  emit("\\vTranslated by PRODUCT" "VERSION" ("REVDATE") on " + stime + "\\v");
}
break;
```

This code is used in section 120.

122. Declarations are output before the start of the body, allowing them to be used to special title generation, if desired. Declarations are an excellent place to define any substitutions to be applied to the subsequent text.

⟨Process declarations in Palm 122⟩ ≡

```
if (bracket ≡ Body) {
  assert(etextBodyParserFilter::isLineSpecial(text));
  if (¬isSubstitution(text)) {
    emit(etextBodyParserFilter::specialCommand(text));
  }
}
break;
```

This code is used in section 120.

123. We save the document title, concatenating into a single line if it spans two or more in the input text. It will eventually be used to declare the document database name and on the title page of the output document.

⟨Process document title in Palm 123⟩ ≡

```
switch (bracket) {
case Begin: htitle = "";
  break;
case Body:
  if (htitle ≠ "") {
    htitle += " ";
  }
  htitle += quotePalmString(pruneIndent(text));
  hastitle = true;
  break;
case Void: hastitle = false;
  htitle = "";
  break;
}
break;
```

This code is used in section 120.

124. The author name is accumulated, concatenating multiple lines as required. When we see the *End* bracket for the author specification (or the *Void* bracket if no author is given), we write the document header. If no document title was specified, the user will have to supply the name of the Palm database when the PML file is compiled into a Palm Reader book.

```

⟨Process author in Palm 124⟩ ≡
  switch (bracket) {
  case Begin: hauthor = "";
    break;
  case Body:
    if (hauthor ≠ "") {
      hauthor += "␣";
    }
    hauthor += quotePalmString(pruneIndent(text));
    break;
  case End: hasauthor = true;
    /* Note fall-through */
  case Void:
    if (hastitle) {
      emit("\\vTITLE=\"" + htitle + "\"\\v");
      emit("\\c\\b" + htitle + "\\b");
      emit("\\c");
    }
    if (hasauthor) {
      emit("\\c" + hauthor);
      emit("\\c");
    }
    break;
  }
  break;

```

This code is used in section 120.

125. We save the chapter number for output after the chapter name is received. The chapter number may span multiple lines.

```

⟨Process chapter number in Palm 125⟩ ≡
  switch (bracket) {
  case Begin: chapnumber = "";
    break;
  case Body:
    if (chapnumber ≠ "") {
      chapnumber += "␣";
    }
    chapnumber += quotePalmString(pruneIndent(text));
    break;
  case Void: chapnumber = "";
    /* Note fall-through */
  case End: break;
  }
  break;

```

This code is used in section 120.

126. Chapter names cause `\x` chapter tags to be generated with the chapter title as its argument. If only a chapter number is given, it is used as the chapter title. If both a number and name are specified, they are concatenated with a colon after the number and the resulting string is used as the chapter title. *Void* chapter names generate chapters numbered $1, 2, \dots, n$.

⟨Process chapter name in Palm 126⟩ ≡

```

switch (bracket) {
  case Begin: chapname = "";
    break;
  case Body:
    if (chapname ≠ "") {
      chapname += "␣";
    }
    chapname += quotePalmString(pruneIndent(text));
    break;
  case Void: chapname = "";
    /* Note fall-through */
  case End: chapno++;
    emit("");
    if ((chapname ≠ "") ∨ (chapnumber ≠ "")) {
      string s = "\\x\\b";
      if (chapnumber ≠ "") {
        s += chapnumber;
        if (chapname ≠ "") {
          s += "␣";
        }
      }
      emit(s + chapname + "\\b\\x");
    }
    else {
      ostreamstream numchap;
      numchap << "\\x\\b\\a151␣" << chapno << "\\a151\\b\\x";
      emit(numchap.str());
    }
    break;
}
break;

```

This code is used in section 120.

127. The *generateFilledParagraph* function handles paragraphs with text which flows from line to line to fill the page. It is used for normal body copy and indented block quotations, which differ only in that the latter are wrapped by `\t` markup tags, passed as the *envtype* argument. Existing indentation on argument lines is discarded, and lines of the paragraph are joined into one line per paragraph as required in PML.

⟨ Class definitions 8 ⟩ +=

```

void PalmGenerationFilter::generateFilledParagraph(bodyState state, string envtype, char
    bracket, string text)
{
    string s;
    switch (bracket) {
    case Begin: emit("");
        quoth = false;
        partext = "";
        break;
    case Body:
        if (etextBodyParserFilter::isLineSpecial(text)) {
            if (!isSubstitution(text)) {
                partext += etextBodyParserFilter::specialCommand(text);
            }
        }
        else {
            s = quotePalmString(pruneIndent(text));
            if (infoot) {
                if (footpar != "") {
                    footpar += '␣';
                }
                footpar += s;
            }
            else {
                if (partext == "") {
                    partext = envtype;
                }
                else {
                    partext += '␣';
                }
                partext += s;
            }
        }
        break;
    case End: emit(partext + envtype);
        break;
    case Void: break;
    }
}

```

128. This function handles the various kinds of aligned paragraphs we encounter in a document. It wraps the contents of the paragraph in a Palm Markup Language environment of the type specified by *envtype*. The indentation used in the input text to identify the alignment of the copy is removed, as indentation is significant in PML. Preformatted tables are a special case; to make the most of limited screen space, we normally strip the two leading spaces present on lines of such tables. If for some strange reason the input document introduces a table with a line which begins in column 3 but a subsequent line of the table contains a nonblank in columns 1 or 2, that line will be output in its entirety. This will misalign the table, but it's better than discarding characters in the input text.

(Class definitions 8) +=

```
void PalmGenerationFilter::generateAlignedParagraph(bodyState state, string envtype, char
    bracket, string text)
{
    string s, l;
    switch (bracket) {
    case Begin: emit("");
        quoth = false;
        parline = 0;
        break;
    case Body: s = "";
        if (parline == 0) {
            s = envtype;
        }
        if (etextBodyParserFilter::isLineSpecial(text)) {
            if (!isSubstitution(text)) {
                s += etextBodyParserFilter::specialCommand(text);
            }
            else {
                break;
            }
        }
        else {
            if (state == InPreformattedTable) {
                l = quotePalmString(text.substr((text.substr(0, 2) == "  " ? 2 : 0)));
            }
            else {
                l = quotePalmString(pruneIndent(text));
            }
            if (infoot) {
                if (footpar != "") {
                    footpar += ' ';
                }
                footpar += l;
                break;
            }
            else {
                s += l;
            }
        }
        emit(s);
        parline++;
        break;
    case End: emit(envtype);
```

```
    break;  
    case Void: break;  
  }  
}
```

129. Translate text string s into PML, quoting metacharacters and expanding Latin-1 characters to decimal escapes. Italic mode, conversion of ASCII quotes to open and close quotes, ellipsis and em-dash translation, mathematics mode, and footnote processing are performed at this level. The handling of footnotes which span multiple lines in the input text interacts in subtle ways with *generateFilledParagraph* and *generateAlignedParagraph*—don't make any structural changes in footnote handling here unless you completely grasp the implications for callers of this function.

⟨ Class definitions 8 ⟩ +≡

```

string PalmGenerationFilter::quotePalmString(string s)
{
    string::iterator cp;
    string o = "";
    int c;
    static const string punctuation = "?!.:;";    /* Punctuation set after space for frenchPunct */
    for (cp = s.begin(); cp < s.end(); cp++) {
        c = (*cp) & #FF;
        if (c < '␣') {
            ⟨ Quote control character in Palm 130 ⟩;
        }
        else if ((c ≥ 160) ∧ (c ≤ 255)) {
            ⟨ Quote ISO 8859-1 character in Palm 131 ⟩;
        }
        else if (c ≥ '␣' ∧ c ≤ '~') {
            if (¬inmath ∧ ¬intable ∧ c ≡ '₋') {
                ⟨ Toggle italic text mode in Palm 132 ⟩;
            }
            else if (¬intable ∧ c ≡ '\\') ∧ ((cp + 1) < s.end()) ∧ ((cp[1] ≡ '(') ∨ (cp[1] ≡ ')')) {
                ⟨ Toggle math mode in Palm 133 ⟩;
            }
            else if (c ≡ '\\') {
                o += "\\\\";
            }
            else if (¬inmath ∧ ¬intable ∧ ((cp + 2) < s.end()) ∧ ((c ≡ '[') ∨ ((c ≡ '␣') ∧ (cp[1] ≡ '[') ∨ (cp[1] ≡ ')')))) {
                ⟨ Begin footnote in Palm 134 ⟩;
            }
            else if (¬inmath ∧ ¬intable ∧ c ≡ ']') {
                ⟨ End footnote in Palm 135 ⟩;
            }
            else if (¬inmath ∧ ¬intable ∧ (c ≡ '-') ∧ ((cp + 1) < s.end()) ∧ (cp[1] ≡ '-')) {
                ⟨ Translate em-dash in Palm 136 ⟩;
            }
            else if (¬inmath ∧ ¬intable ∧ (c ≡ '.') ∧ ((cp + 2) < s.end()) ∧ (cp[1] ≡ '.') ∧ (cp[2] ≡ '.')) {
                ⟨ Translate ellipsis in Palm 137 ⟩;
            }
            else if (¬inmath ∧ ¬intable ∧ c ≡ '"') {
                ⟨ Convert ASCII quotes to open and close quotes in Palm 138 ⟩;
            }
            else {
                ⟨ Output ASCII text character in Palm 139 ⟩;
            }
        }
    }
    /* Note that other characters, specifically those in the range from 127 through 160, get
       dropped. */
}

```

```

    }
    o = transformer.substitute(o);    /* Apply substitutions, if any */
    return o;
}

```

130. This is a control character. Emit as \wedge *letter* unless it is considered as white space (for example, carriage return and line feed), in which case it's sent directly to the output.

⟨Quote control character in Palm 130⟩ ≡

```

    if (isspace(c)) {
        o += c;
    }
    else {
        o += "^";
        o += ('@' + c);
    }
}

```

This code is used in section 129.

131. Palm Markup Language requires that all non-ASCII characters, even those part of the ISO 8859-1 character set, be quoted using the `\axxx` escape sequence. We handle this here. In addition, if *frenchPunct* is enabled, we must check for guillemets and insert the requisite non-breaking spaces to set them off from the text.

⟨Quote ISO 8859-1 character in Palm 131⟩ ≡

```

ostream isochar;
isochar << "\\a" << setw(3) << setfill('0') << c;
if (¬inmath ∧ frenchPunct ∧ (c ≡ C_LEFT_POINTING_DOUBLE_ANGLE_QUOTATION_MARK) ∧ (cp ≠
    s.end()) ∧ (¬isspace((cp[1] & #FF)))) {
    o += isochar.str();
    o += "\\a160";
}
else if (¬inmath ∧ frenchPunct ∧ (c ≡ C_RIGHT_POINTING_DOUBLE_ANGLE_QUOTATION_MARK) ∧ (cp ≠
    s.begin())) {
    o += "\\a160";
    o += isochar.str();
}
else {
    o += isochar.str();
}
}

```

This code is cited in section 158.

This code is used in section 129.

132. The underscore character, “_”, toggles text between the normal roman and italic fonts.

⟨Toggle italic text mode in Palm 132⟩ ≡

```

italics = ¬italics;
if (italics) {
    o += "\\i";
}
else {
    o += "\\i";
}
}

```

This code is used in section 129.

133. PML doesn't support mathematics. When we encounter mathematics in the text, we simply output it as text. The user can, afterward, convert the equation to an image with use `TeXToGIF` or an equivalent tool and insert an image. In the meanwhile, we simply encode the `LATEX` equation to text, quoting special characters as required. Even though we don't directly support mathematics, we need to know when we're in it, since characters such as “_” and “[” are regular text characters, not markup, in math mode.

```
<Toggle math mode in Palm 133> ≡
  inmath = cp[1] ≡ '(';
  o += "\\\\";
```

This code is used in section 129.

134. Footnotes are represented by a number enclosed in square brackets, linked to the footnote at the end of the document, which has a link back to the text. At the start of a footnote we append the footnote mark to the output accumulation string *o*, then save it in *footsave*, setting *infoot* to indicate we're accumulating a footnote. While *infoot* is set, *emit* diverts output to the string *footnotes*, where it is simply concatenated at the end. This string will eventually be appended to the end of the output document when we reach the end of the input text.

We don't allow footnotes to be nested. If the user attempts to nest footnotes, we issue a warning and simply emit the nested footnote in-line (within the outer footnote), enclosed in square brackets.

```
<Begin footnote in Palm 134> ≡
  footnest++;
  if (footnest > 1) {
    issueMessage("Cannot nest footnotes in Palm Markup Language output.");
    o += "[";
  }
  else {
    ostreamstream flink;
    footnum++;
    flink << "\\Q=\"b" << footnum << "\\q=\"#f" << footnum << "\"[" << footnum << "]\\"q";
    o += flink.str();
    footsave = o;
    infoot = true;
    fitalics = italics;
    fquoth = quoth;
    italics = quoth = false;
    o = "";
    footpar = "";
    if (footnum ≡ 1) {
      emit("\\x\\a185\\a178\\a179\\a133\\x"); /* Footnote chapter: “1 2 3...” */
    }
    flink.str("");
    flink << "\\p\\Q=\"f" << footnum << "\"";
    emit(flink.str());
  }
  if ((c ≡ '_') ∧ ((cp + 1) < s.end())) {
    if (cp[1] ≡ '_') {
      cp++;
    }
    cp++;
  }
}
```

This code is used in section 129.

135. Close a footnote when the right bracket is encountered. The footnote paragraph, assembled in *footpar* with the cooperation of the caller of *quotePalmString* if the footnote spans multiple lines, is appended to the *footnotes* array by calling *emit* while *infoot* remains set. Following the footnote a back link to the body copy where the footnote appeared is generated.

⟨End footnote in Palm 135⟩ ≡

```

if (footnest == 0) {
    issueMessage("Mismatched_end_of_footnote_\["\]\_bracket.");
}
else {
    footnest--;
    if (footnest > 0) {
        o += ']'; /* Nested footnote—just emit closing bracket */
    }
    else {
        ostreamstream blink;
        if (o != "") {
            if (footpar != "") {
                footpar += '_';
            }
            footpar += o;
        }
        blink << "\\b" << footnum << ".\\b_";
        emit(blink.str() + footpar);
        blink.str("");
        blink << "\\c\\1\\q=\"#b" << footnum << "\"<<<\\q\\1";
        emit(blink.str());
        emit("\\c");
        infoot = false;
        italics = fitalics;
        quoth = fquoth;
        o = footsave;
    }
}

```

This code is used in section 129.

136. Two adjacent hyphens, “--” denote an *em* dash in an ASCII Etext. Translate this sequence into the em-dash symbol used by PML.

⟨Translate em-dash in Palm 136⟩ ≡

```

o += "\\a151";
cp++;

```

This code is used in section 129.

137. Three consecutive periods are translated into a Palm ellipsis character.

⟨Translate ellipsis in Palm 137⟩ ≡

```

o += "\\a133";
cp += 2;

```

This code is used in section 129.

138. ASCII quote characters are translated into open and close quote symbols. Note that the flag *quoth* is unconditionally reset at the end of a paragraph so that mismatched quotes won't propagate beyond one paragraph. This allows continued quotes in multiple paragraphs to work properly. We also save and restore *quoth* around footnotes so quote matching works when a footnote appears within quotes.

⟨ Convert ASCII quotes to open and close quotes in Palm 138 ⟩ ≡

```
o += quoth ? "\\a148" : "\\a147";
quoth = ¬quoth;
```

This code is used in section 129.

139. Output a text character. Some Palm Markup Language metacharacters require backslash quoting in any mode, others only when not in math mode. PML specifies that only a single space appear after punctuation; we suppress multiple spaces here except when generating a preformatted table.

⟨ Output ASCII text character in Palm 139 ⟩ ≡

```
if (¬inmath ∧ frenchPunct ∧ (punctuation.find_first_of(c) ≠ string::npos) ∧ (((cp+1) ≡ s.end()) ∨ (cp[1] ≡
    '␣') ∨ ((cp[1] & #FF) ≡ C_RIGHT_POINTING_DOUBLE_ANGLE_QUOTATION_MARK) ∨ (cp[1] ≡ ', '))) {
    o += "\\a160";
    o += c;
}
else {
    if (intable ∨ (c ≠ '␣') ∨ (o ≡ "") ∨ (o[o.length() - 1] ≠ '␣')) {
        o += c;
    }
}
```

This code is cited in section 158.

This code is used in section 129.

140. Special commands which define text substitutions are recognised by this method, parsed, and added to the substitution list.

⟨ Class definitions 8 ⟩ +≡

```

bool PalmGenerationFilter::isSubstitution(string cmd)
{
    string s = etextBodyParserFilter::specialCommand(cmd);
    bool isSub = false, bogus = true;
    unsigned int n, m, l;
    char delim;
    if (s.find("Substitute_") == 0) {
        isSub = true;
        s = s.substr(11);
        n = s.find_first_not_of(' ');
        if (n != string::npos) {
            delim = s[n];
            m = s.find(delim, n + 1);
            if (m != string::npos) {
                l = s.find(delim, m + 1);
                if (l != string::npos) {
                    bogus = false;
                    transformer.addSubstitution(s.substr(n + 1, (m - n) - 1), s.substr(m + 1, (l - m) - 1));
                }
            }
        }
    }
    if (bogus) {
        issueMessage("Invalid_Palm_Substitute_special", cerr);
        issueMessage(auditFilter::quoteArbitraryString(cmd), cerr);
    }
}
return isSub;
}

```

141. Main program.

The **etset** program is a filter which processes both its input and output in a strictly serial fashion, permitting it to be used as part of a pipeline. (The program does need to look ahead, but handles this internally.)

⟨Main program 141⟩ ≡

```

int main(int argc, char *argv[]) { int i, f = 0, opt;
    char *cp;
    ⟨Process command-line options 148⟩;
    ⟨Parse command-line file arguments 150⟩;

    streamSource insource;
    trimFilter tfilt;
    tabExpanderFilter tabf(8);
    flattenISOCharactersFilter *fiso;
    convertForeignCharacterSetToISOFilter *dosconv;
    auditFilter afilt(FormatWidth);
    sectionSeparatorSquid squiddley;
    etextBodyParserFilter bodyParser;
    stripSpecialCommandsFilter *ssc;
    LaTeXGenerationFilter *lf;
    PalmGenerationFilter *pf;
    streamSink *os;
    heatSink *hs;
    HTMLGenerationSink *hgs;
#define Plumb(component) *pipeEnd | component; pipeEnd = &component
    try {
        insource.openFile(infile);
    }
    catch(invalid_argument &e)
    {
        cerr << e.what() << "\n";
        return 2;
    }

    textComponent *pipeEnd = &insource;    /* Pipeline begins with input file source */
    if (dosCharacters) {
        insource.setStripEOL(true);
        dosconv = new convertForeignCharacterSetToISOFilter(cp850_to_ISO);
        Plumb(*dosconv);
    }
    if (¬checkText) {
        Plumb(tfilt);    /* Trim trailing white space... */
        Plumb(tabf);    /* ...and expand tabs to spaces. */
    }
    if (specialStrip) {
        ssc = new stripSpecialCommandsFilter;
        Plumb(*ssc);
    }
    if (flattenISOchars) {
        fiso = new flattenISOCharactersFilter;
        Plumb(*fiso);
    }
    if (cleanText ∨ checkText) {

```

```

    afilt.setAuditCriteria(auditFilter :: trailing_blanks | auditFilter :: embedded_tabs |
        auditFilter :: exceeds_maximum_length | auditFilter :: invalid_characters |
        auditFilter :: special_commands_present | (asciiOnly ? 0 :
        auditFilter :: permit_8_bit_ISO_characters));
    Plumb(afilt);
    if (checkText) {
        hs = new heatSink;
        Plumb(*hs);
    }
    else {
        os = new streamSink(outfile);
        Plumb(*os);
    }
}
else {
    Plumb(squiddley);    /* ...and split the input file into sections. */
    { Configure prologue and epilogue processing 142 };
    if (asciiOnly) {
        afilt.disableAuditCriteria(auditFilter :: permit_8_bit_ISO_characters);
    }
    Plumb(afilt);    /* The Etext body section is audited for errors, */
    afilt.disableAuditCriteria(auditFilter :: special_commands_present);
    /* permitting special commands, */
    Plumb(bodyParser);    /* then fed to the body parser. */
    { Set up parser debugging if requested 143 };
    if (ofmt == LaTeX) {
        lf = new LaTeXGenerationFilter;
        os = new streamSink(outfile);
        bodyParser.setSpecialFilter("LaTeX");
        Plumb(*lf);
        Plumb(*os);
    }
    else if (ofmt == HTML) {
        hgs = new HTMLGenerationSink(outfile, singleFileHTML);
        bodyParser.setSpecialFilter("HTML");
        Plumb(*hgs);
    }
    else if (ofmt == Palm) {
        pf = new PalmGenerationFilter;
        os = new streamSink(outfile);
        bodyParser.setSpecialFilter("Palm");
        Plumb(*pf);
        Plumb(*os);
    }
}
insource.send();
if (verbose) {
    cerr << insource.getSourceLineNumber() << "input_lines_processed.\n";
}
return 0; }

```

This code is used in section 6.

142. The prologue and epilogue of the input file are usually discarded, with only the body of the Etext being processed. The user can, by specifying the `--save-prologue` and/or `--save-epilogue` options, each of which takes a file name argument, direct these portions of the input to the designated file. The same file name may be specified for both the prologue and epilogue: the **sectionSeparatorSquid** goes to great pains to ensure this will work.

```
< Configure prologue and epilogue processing 142 > ≡
textComponent *prodest = Λ;
if (savePrologueFile ≠ "") {
    squiddley.setPrologueProcessor(prodest = new streamSink(savePrologueFile));
}
if (saveEpilogueFile ≠ "") {
    if (savePrologueFile ≡ saveEpilogueFile) {
        squiddley.setEpilogueProcessor(prodest);
    }
    else {
        squiddley.setEpilogueProcessor(new streamSink(saveEpilogueFile));
    }
}
```

This code is used in section 141.

143. If the `--debug-parser` option is set, we insert a **teeSquid** into the pipeline after the **etextBodyParserFilter** with its secondary output directed to a **parserDiagnosticFilter** which is in turn plumbed to a **streamSink** which writes the parser diagnostic information on the *debugParserFile* given as the argument to the option.

```
< Set up parser debugging if requested 143 > ≡
if (debugParser) {
    parserDiagnosticFilter *pd = new parserDiagnosticFilter;
    streamSink *pdsink = new streamSink(debugParserFile);
    teeSquid *pdtsq = new teeSquid(pd);
    *pd | *pdsink;
    Plumb(*pdtsq);
}
```

This code is used in section 141.

144. Application plumbing.

Every application needs a modicum of clanking machinery beneath the waterline to get its job done and conform to contemporary community standards. I've relegated these gory and boring details to the end, where you're most sincerely encouraged to ignore them.

145. The following include files provide access to system and library components.

⟨System include files 145⟩ ≡

```
#include "config.h"
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <sstream>
#include <cstdlib>
#include <exception>
#include <stdexcept>
#include <string>
#include <vector>
#include <queue>
#include <map>
#include <algorithm>
    using namespace std;
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <time.h>
#include <assert.h>
#ifdef HAVE_STAT
#include <sys/stat.h>
#endif
#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif
#include "getopt.h"    /* Use our own getopt, which supports getopt_long */
```

This code is used in section 5.

146. Here are the global variables we use to keep track of command line options.

⟨Global variables 48⟩ +=

```

typedef enum {
    LaTeX, HTML, Palm
} outputFormat;
static outputFormat ofmt = LaTeX;    /* Output format */
static bool asciiOnly = false;    /* Permit only 7-bit ASCII in input */
static bool babelon = false;    /* Use LATEX babel package */
static string babelang;    /* Language specification for babel */
static bool singleFileHTML = false;    /* Generate single file for HTML output */
static bool debugParser = false;    /* Generate debug output from body parser ? */
static bool dosCharacters = false;    /* Translate MS-DOS characters to ISO ? */
static string debugParserFile = "";    /* Log file for parser debugging output */
static bool flattenISOchars = false;    /* Flatten ISO 8859-1 8-bit codes to ASCII */
static bool frenchPunct = false;    /* Use nonbreaking spaces for French punctuation */
static string savePrologueFile = "";    /* File to save prologue */
static string saveEpilogueFile = "";    /* File to save epilogue */
static bool specialStrip = false;    /* Strip special commands */
static bool cleanText = false;    /* Clean text for shipment (de-tab, trim trailing spaces) */
static bool checkText = false;    /* Check text for shipment */
static bool verbose = false;    /* Print verbose processing information */

```

147. Procedure *usage* prints how-to-call information. This serves as a reference for the option processing code which follows. Don't forget to update *usage* when you add an option!

⟨Global functions 147⟩ ≡

```
static void usage(void)
{
    cout << PRODUCT << "  --  Typeset ISO 8859 Latin-1 Etext.  Call\n";
    cout << "  with  " << PRODUCT << " [input [output]]\n";
    cout << "\n";
    cout << "Options:\n";
    cout << "  --ascii-only  Permit only 7-bit ASCII characters in input\n";
    cout << "  --babel_lang  Use LaTeX babel package for lang\n";
    cout << "  --check  Check text for publication\n";
    cout << "  --clean  Clean: expand tabs, remove trailing white space\n";
    cout << "  --copyright  Print copyright information\n";
    cout << "  --debug-parser_file  Write parser debugging log to file\n";
    cout << "  --dos-characters  Translate MS-DOS characters to ISO 8859\n";
    cout << "  --flatten-iso  Flatten ISO 8859-1 8-bit codes to ASCII\n";
    cout << "  --french-punctuation  Use nonbreaking spaces for French punctuation\n";
    cout << "  --help, -u  Print this message\n";
    cout << "  --html, -h  Generate HTML\n";
    cout << "  --latex, -l  Generate LaTeX\n";
    cout << "  --palm, -p  Generate Palm REGISTERED_SIGN Reader document\n";
    cout << "  --save-epilogue_file  Save epilogue in file\n";
    cout << "  --save-prologue_file  Save prologue in file\n";
    cout << "  --single-file  Single file for HTML output\n";
    cout << "  --special-strip  Strip format-specific special commands\n";
    cout << "  --verbose, -v  Print processing information\n";
    cout << "  --version  Print version number\n";
    cout << "\n";
    cout << "by John Walker\n";
    cout << "http://www.fourmilab.ch/\n";
}
```

This code is used in section 6.

148. We use *getopt.long* to process command line options. This permits aggregation of single letter options without arguments and both *-darg* and *-d arg* syntax. Long options, preceded by *--*, are provided as alternatives for all single letter options and are used exclusively for less frequently used facilities.

⟨Process command-line options 148⟩ ≡

```
static const struct option long_options[] = {
    {"ascii-only", 0, A, 210},
    {"babel", 1, A, 202},
    {"check", 0, A, 209},
    {"clean", 0, A, 208},
    {"copyright", 0, A, 200},
    {"debug-parser", 1, A, 205},
    {"dos-characters", 0, A, 213},
    {"flatten-iso", 0, A, 212},
    {"french-punctuation", 0, A, 203},
    {"help", 0, A, 'u'},
    {"html", 0, A, 'h'},
    {"latex", 0, A, 'l'},
    {"palm", 0, A, 'p'},
    {"save-epilogue", 1, A, 206},
    {"save-prologue", 1, A, 207},
    {"single-file", 0, A, 204},
    {"special-strip", 0, A, 211},
    {"verbose", 0, A, 'v'},
    {"version", 0, A, 201},
    {0, 0, 0, 0}
};
int option_index = 0;
while ((opt = getopt_long(argc, argv, "hlpuv", long_options, &option_index)) != -1) {
    switch (opt) {
        case 210: /* --ascii-only Permit only 7-bit ASCII characters in input */
            asciiOnly = true;
            break;
        case 202: /* --babel language Use babel package with LATEX */
            babelon = true;
            babelang = optarg;
            break;
        case 209: /* --check Check complete text ready for publication */
            checkText = true;
            break;
        case 208: /* --clean Expand tabs, trim trailing white space */
            cleanText = true;
            break;
        case 200: /* --copyright Print copyright information */
            cout << "This program is in the public domain.\n";
            return 0;
        case 205: /* --debug-parser file Write parser debug output file */
            debugParser = true;
            debugParserFile = optarg;
            break;
        case 213: /* --dos-characters Translate MS-DOS character set to ISO 8859-1 */
            dosCharacters = true;
            break;
    }
}
```

```

case 212: /* --flatten-iso Flatten ISO 8859-1 8-bit codes to ASCII */
    flattenISOchars = true;
    break;
case 203: /* --french-punctuation French-style spacing for punctuation */
    frenchPunct = true;
    break;
case 'h': /* -h, --html Generate HTML output */
    ofmt = HTML;
    break;
case 'l': /* -l, --latex Generate LATEX output */
    ofmt = LaTeX;
    break;
case 'p': /* -p, --palm Generate Palm Reader document */
    ofmt = Palm;
    break;
case 206: /* --save-epilogue file Save epilogue in file */
    saveEpilogueFile = optarg;
    break;
case 207: /* --save-prologue file Save prologue in file */
    savePrologueFile = optarg;
    break;
case 204: /* --single-file Single file HTML output */
    singleFileHTML = true;
    break;
case 211: /* --special-strip Strip special commands */
    specialStrip = true;
    break;
case 'u': /* -u, --help Print how-to-call information */
    case '??': usage();
    return 0;
case 'v': /* -v, --verbose Print processing information */
    verbose = true;
    break;
case 201: /* --version Print version information */
    cout << PRODUCT "_" VERSION "\n";
    cout << "Last_revised:_" REVDATE "\n";
    cout << "The_latest_version_is_always_available\n";
    cout << "at_http://www.fourmilab.ch/etexts/etset\n";
    cout << "Please_report_bugs_to_bugs@fourmilab.ch\n";
    return 0;
default:
    cerr << "***Internal_error:_unhandled_case_" << opt << "_in_option_processing.\n";
    return 1;
}
}

```

This code is used in section 141.

149. Some more global variables to keep track of file name arguments on the command line...

⟨Global variables 48⟩ +≡

```

static string infile = "-", /* "-" means standard input or output */
outfile = "-";

```

150. If no file names are specified on the command line, we act as a filter from standard input to standard output. An input and output file name may be specified. For HTML format output, both input and output file names must be given.

```

⟨ Parse command-line file arguments 150 ⟩ =
  for (i = optind; i < argc; i++) {
    cp = argv[i];
    switch (f) {
    case 0: infile = cp;
            f++;
            break;
    case 1: outfile = cp;
            f++;
            break;
    default: cerr << "Too_many_file_names_arguments_specified.\n";
            return 2;
    }
  }
  if ((ofmt ≡ HTML) ∧ ((f < 2) ∨ (outfile ≡ "-"))) {
    cerr << "Must_specify_output_file_name_for_HTML.\n";
    return 2;
  }
  ⟨ Check for input and output files the same 151 ⟩;

```

This code is used in section 141.

151. One of the most common (and disastrous) fat-fingers in invoking this program is specifying the same name for the input and output file. If undetected, the open of the output file will truncate the input file, destroying it. Here we check for this condition and, if it obtains, bail before doing any damage. We don't perform this check for HTML format output, since HTML generates its own file names based on the specified *basename*, and it's implausible that the input file would have the extension *.html*. Obviously, if input or output is standard I/O, we needn't perform this check.

On systems with a Unix-like *stat* function, if the input and output files both exist, we compare the device and inode numbers to check for aliased file names (due to hard or symbolic links, or a specification such as *./zot.txt*).

```
< Check for input and output files the same 151 > ≡
  if ((ofmt ≠ HTML) ∧ (f ≡ 2) ∧ (infile ≠ "-") ∧ (outfile ≠ "-")) {
    bool io_dup = false;
    if (infile ≡ outfile) {
      io_dup = true; /* File names lexically equal */
#ifdef HAVE_STAT
    }
    else {
      struct stat instat, outstat;
      if ((stat(infile.c_str(), &instat) ≡ 0) ∧ (stat(outfile.c_str(),
        &outstat) ≡ 0) ∧ (instat.st_dev ≡ outstat.st_dev) ∧ (instat.st_ino ≡ outstat.st_ino)) {
        io_dup = true;
      }
    }
  }
#ifdef HAVE_STAT
  }
  if (io_dup) {
    cerr << "Input and output may not be the same file.\n";
    return 2;
  }
}
```

This code is used in section 150.

152. Character set definitions and translation tables.

The following sections define the character set used in the program and provide translation tables among various representations used in formats we emit.

153. ISO 8859-1 special characters.

We use the following definitions where ISO 8859-1 characters are required as strings in the program. Most modern compilers have no difficulty with such characters embedded in string literals, but it's surprisingly difficult to arrange for Plain \TeX (as opposed to \LaTeX) to render them correctly. Since \CWEB produces Plain \TeX , the path of least resistance is to use escapes for these characters, which also guarantess the generated documentation will work on even the most basic \TeX installation. Characters are given their Unicode names with spaces and hyphens replaced by underscores. Character defined with single quotes as **char** have named beginning with \C_ .

```
#define REGISTERED_SIGN "\xAE"  
#define C_LEFT_POINTING_DOUBLE_ANGLE_QUOTATION_MARK #AB  
#define C_RIGHT_POINTING_DOUBLE_ANGLE_QUOTATION_MARK #BB  
#define RIGHT_POINTING_DOUBLE_ANGLE_QUOTATION_MARK "\xBB"
```

154. LaTeX representation of ISO graphic characters.

The following table is indexed by ISO codes 161 to 255, and gives the \LaTeX rendering of the ISO character. Using this table permits compatibility with even the oldest \LaTeX versions but, if you're preparing documents in languages which extensively use the ISO character set, it's much wiser to use the `-b language` option to enable the `babel` package for the specified *language*.

⟨ Global variables 48 ⟩ +=

```
static const char *const texform[] = {"!", "\\\rm\\rlap/c", "\\pounds", "$\\otimes$",
  "\\makebox{\\rm\\rlap{\\hspace*{0.07em}\\scriptsize=}}", "$|$", "\\S", "\\{\\}",
  "\\copyright", "\\b{a}", "\\raisebox{0.3ex}{\\tiny$\\ll$~}}", "$\\neg$", "$-$",
  "\\ooalign{\\hfil\\raise.07ex\\hbox{\\sc_r}\\hfil\\crr\\mathhexbox20D}}", "-",
  "$^{\\circ}$", "$\\pm$", "$^2$", "$^3$", "\\prime", "$\\mu$", "\\P", "$\\cdot$", "\\c{}",
  "$^1$", "\\b{o}", "\\raisebox{0.3ex}{\\tiny$\\gg$}}", "\\small_1{1/4$}",
  "\\small_1{1/2$}", "\\small_1{3/4$}", "?", "\\{A}", "\\{A}", "\\^{A}", "\\~{A}",
  "\\{A}", "\\{AA}", "\\{AE}", "\\c{C}", "\\{E}", "\\{E}", "\\^{E}", "\\{E}", "\\{I}",
  "\\{I}", "\\^{I}", "\\{I}", "Eth", "\\{N}", "\\{O}", "\\{O}", "\\^{O}", "\\{O}",
  "\\{O}", "$\\times$", "\\{O}", "\\{U}", "\\{U}", "\\^{U}", "\\{U}", "\\{Y}", "Thorn",
  "\\ss", "\\{a}", "\\{a}", "\\^{a}", "\\~{a}", "\\{a}", "\\{aa}", "\\{ae}", "\\c{c}",
  "\\{e}", "\\{e}", "\\^{e}", "\\{e}", "\\{i}", "\\{i}", "\\^{i}", "\\{i}",
  "eth", "\\{n}", "\\{o}", "\\{o}", "\\^{o}", "\\{o}", "\\{o}", "$\\div$", "\\{o}",
  "\\{u}", "\\{u}", "\\^{u}", "\\{u}", "\\{y}", "thorn", "\\{y}"};
```

155. MS-DOS code page 850 to ISO translation table.

The following table translates characters in the MS-DOS code page 850 set to the ISO 8859-1 set we work with. This table is included from an external file because its comments use ISO characters which are painful to express in Plain T_EX and, in any case, the translation table is of interest only to historians, geeks, and folks chasing bugs therein. The include file defines the array *cp850_to_ISO* which may be used as a translation table with **convertForeignCharacterSetToISOFilter**.

⟨ Global variables 48 ⟩ +≡
#include "cp850.h"

156. Flat 7-bit ASCII approximation of ISO characters.

The following table is indexed by ISO codes 160 to 255, (#A0–#FF) and gives the flat ASCII rendering of each ISO character. For accented characters, these are simply the characters with the accents removed; for more esoteric characters the translations may be rather eccentric.

```

⟨ Global variables 48 ⟩ +=      /* Latin 1/Unicode Hex Description */
static const char *const flattenISO[] = {" ",      /* #A0 Non-breaking space */
"! ",      /* #A1 Spanish open exclamation */
"cents",   /* #A2 Cent sign */
"GBP",     /* #A3 Pounds Sterling */
"$",       /* #A4 Universal currency symbol */
"JPY",     /* #A5 Japanese Yen */
"|",       /* #A6 Broken vertical bar */
"Sec. ",   /* #A7 Section sign */
"¨",       /* #A8 diaeresis */
"(C)",     /* #A9 Copyright */
"a",       /* #AA Spanish feminine ordinal indicator */
"<<",     /* #AB Left pointing guillemet */
"NOT",     /* #AC Logical not */
" ",       /* #AD Soft (discretionary) hyphen */
"(R)",     /* #AE Registered trademark */
"-",       /* #AF Overbar */
"o",       /* #B0 Degree sign */
"+/-",     /* #B1 Plus or minus */
"^2",      /* #B2 Superscript 2 */
"^3",      /* #B3 Superscript 3 */
"´",       /* #B4 Acute accent */
"µ",       /* #B5 Micro sign */
"PP. ",    /* #B6 Paragraph sign */
". ",      /* #B7 Middle dot */
", ",      /* #B8 Spacing cedilla */
"^1",      /* #B9 Superscript 1 */
"o",       /* #BA Spanish masculine ordinal indicator */
">>",     /* #BB Right pointing guillemet */
"1/4",     /* #BC Fraction one quarter */
"1/2",     /* #BD Fraction one half */
"3/4",     /* #BE Fraction three quarters */
"?",       /* #BF Spanish open question */
"A",       /* #C0 Accented capital A grave */
"A",       /* #C1 acute */
"A",       /* #C2 circumflex */
"A",       /* #C3 tilde */
"A",       /* #C4 diaeresis */
"A",       /* #C5 Capital A ring / Angstrom symbol */
"Ae",      /* #C6 Capital Ae */
"C",       /* #C7 Capital C cedilla */
"E",       /* #C8 Accented capital E grave */
"E",       /* #C9 acute */
"E",       /* #CA circumflex */
"E",       /* #CB diaeresis */
"I",       /* #CC Accented capital I grave */
"I",       /* #CD acute */
"I",       /* #CE circumflex */

```

```

"I",      /* #CF diaeresis */
"Th",     /* #D0 Capital Eth */
"N",      /* #D1 Capital N tilde */
"O",      /* #D2 Accented capital O grave */
"O",      /* #D3 acute */
"O",      /* #D4 circumflex */
"O",      /* #D5 tilde */
"O",      /* #D6 diaeresis */
"x",      /* #D7 Multiplication sign */
"O",      /* #D8 Capital O slash */
"U",      /* #D9 Accented capital U grave */
"U",      /* #DA acute */
"U",      /* #DB circumflex */
"U",      /* #DC diaeresis */
"Y",      /* #DD Capital Y acute */
"Th",     /* #DE Capital thorn */
"ss",     /* #DF German small sharp s */
"a",      /* #E0 Accented small a grave */
"a",      /* #E1 acute */
"a",      /* #E2 circumflex */
"a",      /* #E3 tilde */
"a",      /* #E4 diaeresis */
"a",      /* #E5 Small a ring */
"ae",     /* #E6 Small ae */
"c",      /* #E7 Small c cedilla */
"e",      /* #E8 Accented small e grave */
"e",      /* #E9 acute */
"e",      /* #EA circumflex */
"e",      /* #EB diaeresis */
"i",      /* #EC Accented small i grave */
"i",      /* #ED acute */
"i",      /* #EE circumflex */
"i",      /* #EF diaeresis */
"th",     /* #F0 Small eth */
"n",      /* #F1 Small n tilde */
"o",      /* #F2 Accented small o grave */
"o",      /* #F3 acute */
"o",      /* #F4 circumflex */
"o",      /* #F5 tilde */
"o",      /* #F6 diaeresis */
"/",      /* #F7 Division sign */
"o",      /* #F8 Small o slash */
"u",      /* #F9 Accented small u grave */
"u",      /* #FA acute */
"u",      /* #FB circumflex */
"u",      /* #FC diaeresis */
"y",      /* #FD Small y acute */
"th",     /* #FE Small thorn */
"y",      /* #FF Small y diaeresis */
};

```

157. Release history.**Release 1: August 1993**

Initial release, accompanied *De la Terre à la Lune*. Supported \LaTeX output only. The program was called `ETLATEX` in this release.

Release 2: December 1996

Added support for HTML, generating a document tree consisting of an index and one document per chapter with navigation links. Added support for French-style punctuation. This release accompanied the Etext of *Le Tour du Monde en 80 Jours*. The program was renamed `ETSET` as of this release.

Release 2.1: December 1996

Added the ability to “flatten” 8-bit ISO characters to their closest 7-bit ASCII representation for dumb terminals which cannot display accented characters and an option to warn if the Etext contains any non-ISO characters.

Release 3: September 2001

Essentially rewritten as a C++/STL application in Knuth’s “Literate Programming” paradigm.

Added support for Palm Markup Language (PML) output.

HTML output may now be written as a single file containing all chapters as well as a document tree with one chapter per file.

\LaTeX output may now invoke the `babel` package for language-specific formatting and use ISO 8859-1 characters in the \LaTeX input.

Each output format now permits “special” format-specific commands to be embedded in a master Etext. The interpretation of these commands is up to the code which generates each output format; common applications are passing commands transparently through to the output to include images, etc., and substituting strings in the Etext for format-specific encodings which better represent the original document.

Tools for preparers of Etexts are provided including facilities to syntax check documents, strip format-specific special commands, prologues, and epilogues, convert text to canonical form by expanding tab stops and deleting trailing white space, and more.

This release was coincident with, but not embedded in, the Etext of *Autour de la Lune*.

158. Development log.**2001 August 30**

What with commencing the cleanup and debug phase, particularly in the output format specific parts where it's easy to fix something in one format and forget to make corresponding changes in the other, time has come to start a development log. So here goes.

Renamed the flag which controls French-style spacing around punctuation as the more appropriate *frenchPunct* (this is enabled by the `-f` command line option) and fixed numerous bugs in handling it on both the \LaTeX and HTML sides. \LaTeX now uses a smaller font for the nonbreaking space before punctuation. Note that with the new guillemets there is no need for a special case for spacing after an open guillemet in \LaTeX , but HTML must still test for this condition.

Added support for chapter numbers in HTML single file output. Chapter separators may contain a chapter number, chapter name, or neither. If the separator contains only a number or a name, no rule is generated. Otherwise, a rule separates the chapter number and name or suffices as the entire chapter separator.

Added code to handle multi-line title, author, chapter number, and chapter name specifications for HTML format output. This was already implicitly handled for \LaTeX since all of these items wrap the body in a declaration which may span multiple lines.

We don't permit nested *[footnotes]* and, although they actually happen to work in \LaTeX , they don't in HTML. Added a check for this in HTML output which issues a warning for nested footnotes or end footnote brackets when no footnote is open.

Added a check for close footnote bracket with no footnote open in \LaTeX output as well.

Installed a very nice rendering of footnotes in single file HTML as right aligned tables the text which contained the footnote flowed around, then immediately ripped it right out again (actually, disabled on “`#ifdef NETSCRAPE_SUCKS`” because of the old bugeroo which bit us way back in “*Guns in Space*”—any floating object causes Netscape to forget about a running style, with the result that the page margins are lost at the end of the floating object and thereafter. I know of no work-around for this, so I replaced the nice code with ugly code which simply renders the footnote in-line in a smaller sans-serif font with a light yellow background. Even that Netscape can't completely cope with; when it's adding space to justify a line of text, it forgets about the background-color of the font and adds the justification space in the page background colour instead.

2001 August 31

Moved code which creates the GIF navigation buttons into static methods of **HTMLGenerationSink**. *createNavButton* creates a single button, and *createNavButtons* writes the lot. The embedded definitions of the buttons are now static variables local to *createNavButtons*.

Changed definition of button embedded file arrays to “`const`”, occasioning the usual C++ hoo-rah all around the program.

Added `buttons.h`, `config.h`, and `getopt.h` to the dependencies of `etset.c` in `Makefile.in`.

Added a *createNavigationPanel* method to **HTMLGenerationSink** to generate the navigation panel for a chapter. It's called with the numbers of the previous and next chapters (zero if none exist and a greyed-out button should be generated). Note that the look-ahead required to grey out the next button in the last chapter is not yet implemented—at the moment it blindly makes a bad link to a nonexistent chapter.

Integrated the GNU Getopt (`getopt.c`, `getopt.h`, `getopt1.c`) from the `fileutils-4.1` distribution (`lib` directory). This version supports long options with automatic disambiguation. Since it's covered by the GPL, I included a copy of the GPL as `COPYING.GNU` and added a mention of the status of these files to the main `COPYING` statement. Updated the `Makefile.in` to add the new files to the build and release targets.

(The release target is almost certainly out of date in other ways and will need to be reviewed when we're closer to releasing the thing.)

Replaced explicit 8-bit ISO 8859-1 characters in string literals with references to defined constants based on their Unicode name which express them as escaped hexadecimal characters or strings. While most compilers have no problems with such characters in character or string constants, getting Plain \TeX to display them is quite a challenge and won't necessarily work on an old or very basic \TeX installation. Since the case only comes up a few times, the path of least resistance is to quote them and be safe.

Cleaned up some unreferenced variables and signed/unsigned natters identified by a `-Wall` build.

In multi-file HTML documents, the HTML `<title>` of the chapter documents is now the title of the work with the chapter number (if any) appended following a colon. The earlier practice of using the chapter title looked dumb.

Modified chapter title generation for multi-file HTML to use the chapter number, rule, and chapter name logic as used for single file documents. I kept the code separate because it uses a different level of heading and we'll probably want to tweak it further when aesthetical fine tuning is in order.

Output of the document title, author, and chapter numbers and names in HTML documents was not applying HTML translation, only quoting. This caused markup such as italics not to be honoured. I added *translateHTMLString* calls where these items are output to the documents as text. Note that we don't want to store these items in memory in translated form as there are circumstances in which we need them free of HTML mark-up, for example in the `<title>` section of the `<head>` and as arguments in `<meta>` items.

Got rid of the last old-style C I/O in the command line option handler; it's all C++ streams now.

Generation of the document preamble in \LaTeX was getting a tad long to be embedded in the main case statement. I broke it out into its own section.

2001 September 1

Implemented prejustified tables. A prejustified table begins with a line which begins in column 3 and contains at least one sequence of three or more spaces between two nonblank substrings. This, and subsequent lines until the next blank line are to be rendered as-is, in a fixed width font to preserve alignment. Added documentation for this to `etsetfmt.w` and the `adlune.txt` test Etext, and supported it with a `verbatim` environment in \LaTeX and `<pre>` in HTML. Interpretation of control codes within the text are to be suppressed in tables. I think this is correct now in \LaTeX , but it needs more testing in HTML.

Fixed a nasty bug in alignment classification where *isspace()* was considering ISO characters as blank. We need to eliminate all *isspace()* calls and replace with explicit tests for blank (once we know white space has been expanded) or for ASCII white space, being careful not to be fooled by sign-extended characters on machines where `char` is signed.

Changed `Makefile.in` build commands for "configurator" to assume a standard installation of `autoconf` as opposed to the weird environment on Jura. There's no need to specify an explicit path for the macro library unless it's somewhere other than where `autoconf` expects it to be (usually `/usr/share/autoconf`).

Deleted a bunch of excess baggage from `configure.in`.

The `Makefile.in` did not previously distinguish between C and C++ source files. This was no problem until we integrated the GNU *getopt*, which will not build with a C++ compiler as it contains `KR` style function declarations. I added logic to `Makefile.in` to consider files with an extension of `.cc` as C++ and `.c` as C, using the appropriate compiler, and a target to produce a `.cc` file from a `CWEB .w` file by processing it with `CTANGLE` then renaming the resulting `.c` file as `.cc`. (Any additional files produced from the `.w` can specify their extensions explicitly, but this allows using the default output of `CTANGLE`.)

Added logic to `configure.in` to set `PAGER` to `less`, `more`, or `cat`, in that order, depending on which are present.

Converted command line option processing to use *getopt.long* and provided long option alternatives for all existing single letter options. Updated *usage* to document the new long options.

2001 September 2

Added logic to **auditFilter** to detect when lines belong to a preformatted table and permit embedded spaces in them.

Added a check to **auditFilter** which warns when a line judged to be centred by **etextBodyParserFilter::classifyLine** has a discrepancy of more than 2 between the number of spaces at the left and the right (considering the line to extend to *maxLineLength* characters). In addition to ugly centring in the Etext, this will catch many errors in aligning block quotes and ragged left and right text.

Installed logic to dynamically assemble the processing pipeline with dynamically allocated components where required. A *Plumb()* macro in the main program now attaches each component to the end of the pipe, advancing a *pipeEnd* pointer as it goes. You can, of course, still create static pipelines but this approach provides the flexibility we'll need to support all the various task options soon to be provided. For the first time it's actually possible to choose L^AT_EX or HTML format output via the command line options!

Added a **--single-file** option to enable single file HTML output. The default remains a document tree with one file per chapter.

Added a **--debug-parser file** option to enable parser debugging without any compile-time conditionals. If set, a tee is inserted in the pipeline after the body parser with the secondary output directed to a parser diagnostic filter whose output in turn goes to a **streamSink** which writes the parser debug log to the specified *file*.

Added the ability to direct the prologue and epilogue of the input file to designated output files. The **--save-prologue** and **--save-epilogue** options both take a file argument designating where that section of the input is written. These may be the same file name, in which case the prologue and epilogue are concatenated to the same file.

Added the ability to configure **auditFilter** for which tests should be performed on text it processes. A new optional argument to the constructor (default is all tests enabled) accepts a bit map of **enum** type **audit_criteria**, defined public in the class definition. You may also set the criteria with *setAuditCriteria* and return them with *getAuditCriteria*.

Implemented footnote support in multi-file HTML output. When the first footnote is encountered, a footnote document named *basename_foot.html* is created. Each footnote in the document is given an ascending number, which is used as a link in the main document (as a superscript number) which points to a fragment tag in the footnote document. Footnotes are separated by blank lines in a **<pre>** tag so the selected footnote will appear at the top of the page. The footnote document is opened with a target window of *basename_foot*, so the main document continues to be displayed in browsers which support targeted links.

2001 September 3

Added a **--clean** option to perform final cleanup and canonicalisation of a complete Etext prior to publication. This processes the entire input, ignoring section breaks, and expands tabs to spaces, removes trailing white space, and audits the result for trailing blanks and embedded tabs (as an internal sanity check), lines which exceed the maximum length, and invalid characters. The higher-level body-only auditing is not done since the prologue and epilogue need not and usually will not conform to the specifications for the body. Output is to the second argument on the command line or standard output if it is omitted (just like L^AT_EX output), and error messages are directed to standard error.

Added a **--check** option which assumes the input is in the canonical ready-to-publish form produced and verified by the **--clean** option; any text produced by **--clean** with no warnings should produce no warnings when run with **--check**. The text is examined for trailing space, embedded tabs, lines which exceed the maximum length, and invalid characters. There is no output other any warning message, if any, which

are written to standard output. Before publishing a text, it's always a good idea to verify it with `--check`, since it's only too easy when making last minute edits to embed a tab which may mess up formatting on a reader's system with different assumptions about tab stops.

Modified **auditFilter** so lines containing embedded tabs diagnosed when *embedded_tabs* is set in the auditing criteria are not double reported as having invalid characters. Also, all embedded tabs are now reported, not just the first one in the line.

Added a *quoteArbitraryString* function to **auditFilter**, used when printing lines which merit diagnostic messages. This function expands all characters which fail the *isCharacterPermissible* test to their C `\xNN` escape form so non-graphic characters are rendered visible and nondestructive. In addition, blanks appearing at the end of lines are quoted as `\x20` so they're apparent.

Added logic to **HTMLGenerationSink** to cache lines of a chapter in a queue until either the start of the next chapter is encountered or the end of document is reached. Only after we know whether a chapter actually follows the current chapter do we generate the chapter's HTML file. This permits disabling the "next" button for the last chapter, as opposed to allowing the user to click it only to receive a broken link as a reward.

2001 September 4

Added `"const"` to declarations of several read-only tables.

Disabled generation of the footnote navigation button on `"#ifdef FOOTNOTE_BUTTON_NEEDED"`. At the moment we use a superscript footnote number instead of this button.

Implemented support for "special commands". A special command appears in the document body like a regular line of text and obeys the same justification rules. Only one special command may appear on a line, and each as the format:

```
<><><>Special:FORMAT ...Format-specific text...<><><>
```

where *FORMAT* identifies which output format this special command pertains to, for example "HTML" or "LaTeX". By default, **etextBodyParserFilter** strips all special commands from text before passing it downstream. To receive special commands for a given format *FMT*, the *setSpecialFilter* method of the body parser should be used to specify the format desired; special commands for other formats will continue to be elided. To receive all special commands, set the special filter to `"*"`.

Implemented a special command handler in **HTMLGenerationSink** which simply passes through the body of the special to the output HTML file unquoted (this will be the way most format handlers implement specials). This permits including figures in a document with a special like:

```
<><><>Special:HTML <><><>
```

Note that the figure will form part of a paragraph with whatever alignment is specified by the justification of the special. You can, of course, override this by including explicit HTML tags within the body of the special.

Added code to command line parsing to verify that the input and output files aren't the same, which would disastrously truncate the input file before it was even read. If the output format is not HTML, two names are given, and neither is standard I/O, we first test if they're lexically equal and, if not and we're running on a system which supports Unix *stat()*, we test for identical device and Inode numbers to protect against aliased files.

Added the concept of a "Declarations" section consisting entirely of special commands which appear before the title (or first chapter, or whatever—at the top of the document). Declarations begin with the first special command encountered prior to the title and continue until a non-special is encountered. (Note that specials not pertaining to the downstream components are already filtered out before this processing is performed.) The block of declarations is emitted between **Begin** and **End** brackets with each declaration bearing a **Body** bracket in the same manner as any other text block. If multiple sequences of consecutive specials appear,

separated by blank lines, they will be output as multiple declaration blocks, each with its own **Begin** and **End** brackets.

Added support for declarations in **LaTeXGenerationFilter**. Declarations are output in the document preamble, prior to the `\begin{document}` statement. This allows declarations to include packages and make declarations in the global document context. For example:

```
<><><>Special:LaTeX \usepackage{graphics}<><><>
<><><>Special:LaTeX \newcommand{\fig}[1]{\resizebox{8cm}{!}<><><>
<><><>Special:LaTeX {\includegraphics{figures/#1.epsf}}<><><>
```

Note how long specials may be split onto multiple lines as long as the target language permits such syntax.

2001 September 5

Added *enableAuditCriteria* and *disableAuditCriteria* methods to **auditFilter** to set and clear bit masks in the audit criteria mask.

Exempted special commands from line length and improper centring checks in **auditFilter**.

Configured **auditFilter** to permit special commands in texts being translated to \LaTeX and HTML.

Added a `--verbose` (or `-v`) option to enable gabby chatter about what's going on to standard output.

Added `--verbose` mode output to show the number of lines written to a \LaTeX file and each HTML file generated.

Added a *permit_8_bit_ISO_characters* mode to the audit criteria of **auditFilter** (so-phrased so it's enabled if you use the default of “*everything*”), and a new command line option “`--ascii-only`” which clears this mode when building the pipeline. If you require the text to be exclusively 7-bit, you can verify it using this option. The option works in conjunction with the `--check` and `--clean` options as well as those which translate the text.

Added a **stripSpecialCommandsFilter** which removes all special commands from the stream. If elision of special commands would result in two consecutive blank lines, the blank line following the special(s) is deleted as well. A new “`--special-strip`” option interposes this filter after the trailing white space and tab expansion filters (or directly after the input source if these filters are suppressed by the `--check` option). The stripping of specials may then be applied when producing a text for publication with the `--clean` option, or when formatting the text.

2001 September 6

Added `<link>` tags to the header of chapter documents in multiple file HTML to indicate the parent/child relationship between the chapters and the index document and the next/previous relationships among chapters.

Added `<meta>` “`description`” and “`author`” tags to header section of all HTML documents for which a title and author were specified.

Implemented declarations in HTML. Special declarations which appear before the title are saved in a *dqueue* “*declarationsQueue*” and then output prior to the `</head>` tag by *writeHTMLDocumentPreamble*. Declarations are transcribed verbatim to each HTML file generated. You may include as many lines as you wish in the header simply by providing as many consecutive declaration lines.

The check for extra embedded spaces in **auditFilter** should have been suppressed for special commands. It is now.

2001 September 7

Added a `-flatten-iso` option which interposes a **flattenISOCharactersFilter** in the pipeline. This filter translates 8-bit ISO characters to the nearest 7-bit ASCII equivalent, stripping accents from letters and

representing punctuation as best as possible. This filter may be used to extract a flattened Etext with the `--check` option, or to flatten input prior to formatting.

If a document title is specified, it will now appear centred at the top of each HTML chapter document, with the navigation buttons at the right.

If chapter numbers are specified, they will be used to identify chapters in the index document created for multiple file HTML output as the terms in a definition list. Both the terms and the definitions (chapter titles) are linked to the chapter documents. If no chapter number is given, its chapter number ($1 \cdots n$) is used, followed by a period.

The incompatibility brigade having set its sights on the humble `<DL COMPACT>` tag (why would you *need* a list user-supplied items and descriptions appearing on the same line, after all?), I gave up and made the chapter table in the multiple file HTML index document a `<TABLE>`. After sufficient tweaking, it appears to behave reasonably in Netscape and Mozilla. I have yet to subject it to the tender embrace of Explorer.

When generating multiple file HTML, if a special appeared between the title/author and the first chapter number/name item, it would be placed within the table of chapters. This isn't what you want—such specials are likely to be figures at the start of the document or some such, and shouldn't have to conform to the constraints of being embedded in the chapter table. I deferred generating the start of the chapter table until the first chapter title is encountered (at the same time the navigation buttons are written), shifting specials to before the start of the table.

Added a bogus column to the chapter table in the multiple file HTML index document to separate the chapter number from the chapter title.

2001 September 8

Converted this development log into T_EX in a `log.w` file which is included in `etset.w` so it's automatically formatted when the program is printed. I added `log.w` and `etsetfmt.w` as dependencies of `etset.tex` in `Makefile.in`.

2001 September 9

Added documentation of the command line and options to the top of `etset.w`.

Deleted code conditional on `OLD_GUILLEMETS`—the new ones are so much prettier.

Implemented a “Substitution” special command for L^AT_EX. A text line of the form:

```
<><><>Special:LaTeX Substitute /oeil/\oe il/<><><>
```

will substitute the text within the second set of delimiters (which may be any character) for the text within the first, wherever it may occur. Substitution *is not* recursive—substituted text is not re-scanned for occurrences of the same substitution. Note that substitutions are applied in *quoteLaTeXString* after all transformations from the original text to L^AT_EX are applied; this provides the maximum flexibility for overriding the default translation of text.

Absent any widely-available rendering engine for mathematics, we don't translate mathematics into HTML—it is simply output in its L^AT_EX representation. Since editors of HTML documents will usually wish to replace this gnarl with images of the typeset equation produced with tools such as

T_EXtoGIF

I added code to typeset mathematics in pink `<table>` boxes to make them more apparent when proofreading the document.

2001 September 16

Added code to `PalmGenerationFilter::quotePalmString` to quote all non-ASCII characters as `\axxx` escapes; the documentation doesn't make this clear, but it is required unless you fancy a warning message for each and every ISO character.

Fixed an error in generating ellipses for Palm Reader documents; a little left-over code from \LaTeX generation was resulting in the pesky “`Illegal control code: \`” messages.

Added code to re-format paragraphs in Palm Markup Language in “one line per paragraph” as it expects. At the moment only justified paragraphs are so-formatted.

Removed pre-existing indentation before title, author, and chapter titles. Multi-line chapter titles are aggregated onto a single line.

2001 September 17

Palm output wasn’t properly keeping track of math mode, which resulted in \TeX subscript symbols being treated as italic toggles, completely messing up subsequent text. Even though we don’t do anything with math mode, we need to keep track of whether or not we’re in it, as it affects quoting of characters therein.

Added a blank line before chapter breaks in Palm Markup Language output. It doesn’t affect the generated Palm document, but it makes it a lot easier to scan through the PML text when looking for problems.

Pruned existing indentation when generating an aligned paragraph for PML. PML treats indentation as text characters when justifying the line, which messes up the intended alignment. A special case is required for preformatted tables, since we cannot blindly strip significant indentation which follows the two character indentation which marks the table; for tables only the first two blanks are stripped. Note that this behaviour is slightly different from that of \LaTeX and HTML output where the table indentation is preserved and it would be possible to have a table containing lines with nonblank characters in the first or second columns of lines subsequent to the first. While this might be nice, two lost characters are a horrible price to pay on the cramped Palm screen which has difficulty fitting aligned tables of any kind, so as a compromise I strip the first two characters of any line so long as they are blank, but preserve the leading two characters if either is nonblank. Yes, this may misalign a table, but in the vast majority of cases it will provide the best rendering of the author’s intent. Since Palm Reader doesn’t presently support a monospaced font option, there’s no hope of properly aligning preformatted tables in any case—the author is going to have to do it by hand with the `\T=` tag after the PML is generated.

Tweaked *generateAlignedParagraph* for Palm output to cope with an eccentricity of PML paragraph justification tags. The tags ending centred and right justified paragraphs must appear at the start of a line, but the tag ending an intended block quote must be at the *end of the last line*. If you place it at the start of a new line, you end up with an extra line after the paragraph. We now cope with this, buffering text one line ahead in the case of a block quote so the closing tag may be appended when the *End* bracket is encountered.

Note: Palm **MakeBook** and **DropBook** don’t like chapter titles which exceed 80 characters, and issue a warning (or “error” in the case of **MakeBook**) which suggests that chapter markers may be unpaired. The chapter title is, nonetheless, properly rendered in the document and truncated with an ellipsis in the go to chapter form. (Note that chapter titles much shorter than 80 character will usually be truncated in this form as well.) I leave such titles intact, since it’s better to put up with a warning than lop them off or include some kind of kludge which would render poorly when the document is read.

One more little twist with block quotes: naturally they too should be output one paragraph per line with a `\t` before and after; I diddled the code to accomplish this, which actually simplified it. I may make this a special case of generating a justified text paragraph rather than *generateAlignedParagraph*, since the logic now more closely resembles the former.

Made the *pruneIndent*, *elideNewLines*, and *linesIn* private helper methods of **HTMLGenerationSink** **static**; they don’t need no steenkin instance variables.

Implemented special commands and declarations for Palm output, including “**Substitute**” specials. Non-substitution specials emit their text directly into the text being assembled and hence will be embedded in the middle of a paragraph if they appear in regular text or a block quote. Declarations are output before the start of the text and may be used for special titling.

Spacing around guillemets in *frenchPunct* mode was broken due to yet another signed/unsigned **char** problem with the ISO guillemets which are, of course, negative when treated as a **signed char**. I changed the definition to hexadecimal constants and added **char** quantities which may be signed with **#FF** where appropriate to guarantee the comparison will be valid. This required corresponding changes in the *frenchPunct* handling of L^AT_EX and HTML generation as well. Since we need to quote all ISO characters for the Palm, spacing around guillemets had to be handled separately in the `<Quote ISO 8859-1 character in Palm 131>` handler. This actually simplified the code, since the case of punctuation followed by a right guillemet is more clearly distinguished from guillemet handling itself.

Palm markup language prescribes a single space after punctuation, not two. I modified `accrual` in `<Output ASCII text character in Palm 139>` to check whether the last character in the string being assembled is blank, if so the space is discarded. Note that this occurs when quoting document text, so you can still insert multiple spaces in a special, should you need to. Naturally, we don't do this within a preformatted table.

Ripped out the *terminator* argument from *generateAlignedParagraph* for the Palm—it was no longer used.

Replaced the old logic for generating body paragraphs with a new *generateFilledParagraph* method which takes the same arguments and works in the same fashion as *generateAlignedParagraph*, but joins the lines of the paragraph into a single line with optional markup tags at the start and end. This is now used to generate body paragraphs (with null markup tags) and indented block quotes with `\t` tags. This allowed me to eliminate all the special cases for block quotes from *generateAlignedParagraph*, dramatically simplifying it.

2001 September 18

Modified `Makefile.in`'s `clean` target to delete `*.pml` files left around from testing.

Added a “**reconfigure**” target to `Makefile.in` to facilitate testing on multiple platforms. It deletes `config.cache`, re-runs `./configure`, then does a “**make clean**” using the newly-generated `Makefile`.

Split L^AT_EX, HTML, and Palm generation into separate `latex.w`, `html.w`, and `palm.w` CWEB files, all included in the main `etset.w` with the `@i` control code. This facilitates comparing code among formats since each can be opened in a separate window as opposed to forever scrolling back and forth.

Had another go at persuading the `Makefile` to comprehend the fact that while CTANGLE writes a `.c` file, we want to compile it as a `.cc` file. The last attempt would fail in a bizarre way if CTANGLE detected an error and exited, leaving a `.c` file around, which the next `make` would attempt to compile with the C compiler instead of C++.

Added a check to **auditFilter**, governed by audit criterion *trailing-hyphen*, which checks for the common sin in scanned documents of a hyphenated word which the editor has forgotten to join in the Etext. (Note that each occurrence of a hyphen at the end of a line must be reviewed by the editor to determine whether the hyphen was inserted between syllables or existed before the word was broken, as for example in “Franco-Prussian”.) A trailing hyphen is reported only if it is preceded by a letter, including ISO accented letters. A public **static** function, *isISOletter* is provided by **auditFilter** to other code which may need to determine if a character is an ISO letter.

2001 September 19

Spent all day implementing footnotes in Palm Markup Language output. Well...to be more precise, I spent about 15 minutes designing, implementing, and debugging how I intended footnotes to work, then the rest of the day psychoanalysing DropBook (version 1.1.1) and bugs in Palm Reader (version 1.0.6), both versions being the latest released as of this writing.

Footnotes in PML documents, like those in HTML, may not be nested—if nested footnotes appear in the input text a warning is issued and the nested footnotes are simply included in the outermost footnote surrounded by square brackets, just as in the input. When a footnote appears, a link is placed in the output string being assembled by *quotePalmString* and a link is inserted, consisting of the footnote number enclosed

in square brackets, with link destination “*f**n*” where *n* is the footnote number. An anchor named “*b**n*” is placed before the footnote link in the text; this permits returning from the footnote to the text in which it appears.

Footnotes are placed in a pseudo-chapter added to the end of the document; to avoid language-specific nomenclature, this chapter is named “^{1 2 3 ...}”. A page break appears before each footnote in this chapter, and each footnote begins with its number and a period in bold type. At the end of the footnote text is a link back to the body where the footnote appeared; the target of this link is the bold string “<<<”, once again avoiding language-specificity. In case the clever notion of using guillemets for such a link pops into your head, invite it to pop right back out—Palm Reader goes into gibber gibber land if it sees an ISO character in a link target and starts scribbling all over system and unallocated memory. Maybe they’ll fix that some day.

When the opening bracket of a footnote is encountered, the current encoded string being assembled by *quotePalmString* is saved in *footsave* and the flag *infoot* is set. The text processing modes *quoth* and *italics* are also saved and reset to their defaults at the start of a paragraph.

You’ll recall that *generateFilledParagraph* must concatenate lines into one monster line per paragraph or else Palm Reader will faithfully start a new paragraph at each end of line, resulting in horrid looking text. Consequently, it can’t call *emit* for each line of the body it receives, but must assemble lines into the single line paragraph. As a result, it must also handle strings returned by *quotePalmString* with *infoot* set, concatenating them itself into the *footpar* being assembled. Even though *generateAlignedParagraph* usually emits quoted lines as they are completed, it still must assemble footnotes into a single line per footnote (we assume each footnote is a single paragraph) so that lines will flow when it is displayed. Thus, when *infoot* is set it concatenates footnote text returned by *quotePalmString* to *footpar*.

When the right bracket delimiting the end of a footnote (only the outermost, if they are nested) is encountered, the text assembled so far for this line is concatenated to that from earlier lines, if any, and the result, with footnote number, anchor, and link back to the text where the footnote appeared is output, using *emit* to append it to the master *footnotes* string. The partial translated string saved at the start of the footnote is then restored, along with the text modes in effect at that time, and processing of normal text resumes.

2001 September 20

Bugged out last night without ever describing how footnotes actually make it into the PML output file. When the closing bracket of a footnote is processed, the anchor and text accumulated in *footpar* is output by calling the class-local version of *emit* which, with *infoot* set, appends its argument plus a new line to the string *footnotes* rather than passing it down the pipeline. The back link from the footnote to the text is similarly appended by calling *emit*. Finally, when the end of input is reached and we receive the *EndOfText* item, all that need be done is to call *emit(footnotes)* with *infoot false*, which passes all the accumulated footnotes down the pipeline.

Document title and author specifications which spanned multiple lines did not work in PML. Integrated code from **HTMLGenerationSink** to collect them into a single line as required in PML.

Added logic to **HTMLGenerationSink** to push the current italic text state when a footnote is encountered and pop it at the end of the footnote (if footnotes are [improperly] nested, this applies only to the outermost footnote). Also, nested footnotes are now output like in-line footnotes in single file HTML, rather than simply being merged with the text.

A footnote which appeared in a centred, ragged right, or ragged left paragraph in multiple file HTML output would preserve the line breaks in the input text in the footnote document. I modified **HTMLGenerationSink::**
generateAlignedParagraph to skip appending the *terminator* when *infoot* is set upon return from *translateHTMLString*.

C++ “mountain range” identifiers like:

```
pneumaticJackHammer::diggaDiggadig
```

can wreak havoc with \TeX line filling, and the **CWEB** macros don’t honour either a “@|” optional line break in the middle of one or even two adjacent “|” constructs separated by a space. I defined a \TeX macro

“\breakOK” to use within such items (usually after the double colon, between two adjacent C items), to permit them to be broken more aesthetically.

Defined T_EX macros \atsign “@”, \bslash “\”, \caret “^”, \uline “_”, and \vbar “|” to make it easier to talk about such characters in this document without clever special-case quoting.

Here are some things to watch out for when creating PML documents with embedded images. First of all, the images must be created in PNG format and placed in a subdirectory of the directory containing the .pml file you’re compiling. If your PML file is /home/elvis/palmdoc/hounddog.pml then the images must be placed in a directory named /home/elvis/palmdoc/hounddog_img. The PNG files for these images must be of the “palette” type. As of DropBook 1.1.1, grey scale and other types of PNG files do not work. To determine which kind of PNG file and how big it is, if you have the NETPBM and PNG utilities on your system, use the command:

```
pngtopnm name.png |_pnmfile
```

This will produce output like:

```
pngtopnm: reading a 112 x 158 image, 8 bits palette
pngtopnm: writing a PGM file (maxval=255)
stdin:  PGM raw, 112 by 158  maxval 255
```

If the first line does not indicate “8 bits palette”, you’re probably in for trouble; you’ll need to load the image with an image editing program and convert it to an 8-bit palette image. Take note of the image size as well. An image of 158 × 148 pixels will display in-line in the document, while a larger image will be represented by an icon the user must tap to display the actual image. If the image is larger than 158 × 158 pixels the user will be required to scroll the screen to see the complete image. Finally, the actual PNG file embedded in the document may be no larger than 65505 bytes, as this is the maximum size of the Palm database records in which they are stored. If your image is larger than this, you’ll need to reduce resolution and/or select compression modes which reduce it to 65505 bytes or smaller. None of these constraints have anything to do with this program proper; I mention them here in the interest of sparing you some of the frustrations I experienced trying to make illustrated PML documents while testing it.

2001 September 21

Palm output didn’t show the number of lines of PML generated when the “--verbose” option was specified; fixed. One subtlety is that when the footnotes are appended to the end of the document, *emit* is called with a line which may contain one or more embedded end-of-line characters. This is a little shoddy, but it works just fine and saves us from all the complexity of a line queue and separate calls on *emit* whose only benefit would be purity of essence. It does mean, however, that in order to accurately count the line written to the PML file we need to count the number of new line characters in the aggregate footnote string and add that to the number of lines counted by *emit*; this is easily accomplished.

Cleaned up formatting of the input syntax documentation in `etsetfmt.w`.

Plain C++ `iterator` was not defined in `cweb/c++lib.w` as a type name; I added it.

2001 September 22

To make it easier to cope with Project Gutenberg source documents which are perversely published in MS-DOS Code Page 850 8-bit characters, I added a `convertForeignCharacterSetToISOFilter` which, driven by a translation table, converts characters in the lines it passes through. I defined a Code Page 850 to ISO translation table in the file `cp850.h`, which is included in `etset.w`. (Keeping the conversion table in a separate file allows me to use ISO characters in comments without gnarly encoding for CWEB in a table which nobody will ever look at anyway.)

2001 September 23

Added a `--dos-characters` command line option to place a `convertForeignCharacterSetToISOFilter` immediately after the stream source at the head of the pipeline. The filter uses the *cp850_to_ISO* translation table to convert DOS characters to ISO 8859.

Added the ability to strip DOS carriage returns from the ends of lines in `streamSource`. A new *setStripEOL* method, called with an argument of *true*, enables carriage return stripping. Only a single carriage return will be stripped from the end of lines, and lines which do not contain a trailing carriage return will not be modified. This mode is activated by the `--dos-characters` option. There's a *getStripEOL* method to inquire if stripping is enabled in case you need to know.

Added `cp850.h` to dependencies of `etset.cc` in `Makefile.in`.

Fixed a few more instances of awkward grammar in `etsetfmt.w` in the process of integrating the text into the latest version of *Autour de la Lune*.

Brought the README up to date.

2001 September 24

Integrated current description of command line options and input syntax into the manual page `etset.1`.

Placed the Web document for the program in subdirectory `webdoc` and updated.

Added a `check` target to `Makefile.in` and a `reference` target to rebuild the `check_master.txt.gz` file which the output of the check run is compared against.

Made a `makew32.bat` file to build the Win32 executable with DJGpp (compiling `getopt.c` and `getopt1.c` with `gcc`). Added a `testw32.bat` file to semi-automate testing on Win32. You still have to do the `diff` by hand, since there aren't stock utilities to perform this step on Win32.

2001 September 25

Cleaned up some signed/unsigned natters from a `-Wall` build in code related to special command parsing and processing.

Added null destructors declared `virtual` to `LaTeXGenerationFilter` and `PalmGenerationFilter` to get rid of natters about "class has virtual functions but non-virtual destructor".

Fixed a rather subtle signed/unsigned problem which has bedeviling me on the Win32 build for the last day. Consider code of the form:

```
string s;
unsigned int i;
for (i = s.length() - 1; i >= 0; i--) {
```

where you wish to scan a string in reverse order for something or other. If *s* is the null string, the expression *s.length()-1* will go negative, which will be treated as a huge positive value since *i* is **unsigned**. What this does is architecture- and compiler-dependent; on the Win32 build with DJGpp, it appears to have indexed off the start of the string, which caused the code in (Check for line with trailing white space 53) to randomly (actually non-repeatably) report trailing blanks on lines which were actually empty. I changed the iteration variable to a regular **int** and the problem went away.

Modified the `dist` target in `Makefile.in` to handle subdirectories (such as `cweb` and `webdoc`) included in the distribution.

Added dependencies to the `dist` target to ensure the PDF documentation and C++ source are current.

How embarrassing! In testing distribution archives, I discovered that the program did not detect if its input file did not exist. To handle this in a thoroughly clean manner, I broke out the file open code in `streamSource` into a separate *openFile* method which throws an **invalid_argument** argument if the file does not exist. If you wish to catch this exception, construct the `streamSource` with no arguments (initially

binding it to *cin*), then perform the *openFile* within a **try** block which handles the exception that results if the input file does not exist.

2001 September 26

Added `test.txt` to the distribution. It was its absence which provoked yesterday's alarums. Also needed to include `check_master.txt.gz` in the archive to run `check` tests on builds.

Corrected dependencies in `Makefile.in` so generation of `config.h` by `configure` won't require `etset.cc` to be regenerated from the `.w` files; `etset.o` depends on `config.h`, but `etset.cc` doesn't.

2001 September 28

Corrected an error advancing past the opening bracket of a footnote in *quotePalmString*; a footnote which followed the two spaces at the end of the sentence would incorrectly be reported as nested.

159. Index. The following is a cross-reference table for **etset**. Single-character identifiers are not indexed, nor are reserved words. Underlined entries indicate where an identifier was declared.

addSubstitution: [66](#), [90](#), [140](#).
afilt: [141](#).
alignment: [91](#), [102](#).
argc: [141](#), [148](#), [150](#).
argv: [141](#), [148](#), [150](#).
asciiOnly: [141](#), [146](#), [148](#).
assert: [8](#), [19](#), [24](#), [45](#), [52](#), [71](#), [91](#), [93](#), [119](#), [122](#).
audit_criteria: [52](#), [158](#).
auditFilter: [16](#), [52](#), [62](#), [63](#), [90](#), [140](#), [141](#), [158](#).
Author: [31](#), [33](#), [34](#), [35](#), [48](#), [69](#), [92](#), [120](#).
babelang: [70](#), [146](#), [148](#).
babelon: [70](#), [79](#), [146](#), [148](#).
basename: [91](#), [92](#), [98](#), [100](#), [101](#), [108](#), [110](#), [115](#).
BeforeTitle: [30](#), [31](#), [32](#), [48](#).
Begin: [31](#), [34](#), [36](#), [40](#), [47](#), [49](#), [69](#), [72](#), [73](#), [74](#), [75](#), [76](#), [94](#), [95](#), [97](#), [98](#), [102](#), [120](#), [123](#), [124](#), [125](#), [126](#), [127](#), [128](#).
begin: [20](#), [22](#), [23](#), [63](#), [67](#), [77](#), [91](#), [103](#), [105](#), [112](#), [120](#), [129](#), [131](#).
BeginText: [28](#), [30](#), [31](#), [48](#), [69](#), [92](#), [120](#).
BetweenParagraphs: [31](#), [33](#), [34](#), [35](#), [36](#), [37](#), [38](#), [39](#), [40](#), [41](#), [42](#), [48](#), [59](#).
bf: [117](#).
binary: [117](#).
blink: [135](#).
bname: [91](#).
Body: [32](#), [34](#), [35](#), [36](#), [37](#), [38](#), [40](#), [41](#), [47](#), [49](#), [71](#), [72](#), [73](#), [74](#), [75](#), [76](#), [93](#), [94](#), [95](#), [97](#), [98](#), [102](#), [122](#), [123](#), [124](#), [125](#), [126](#), [127](#), [128](#).
bodyParser: [141](#).
bodyStartLines: [113](#).
bodyState: [28](#), [42](#), [47](#), [48](#), [50](#), [52](#), [64](#), [69](#), [92](#), [119](#), [120](#), [127](#), [128](#).
bogus: [90](#), [140](#).
bracket: [28](#), [68](#), [69](#), [71](#), [72](#), [73](#), [74](#), [75](#), [76](#), [91](#), [92](#), [93](#), [94](#), [95](#), [97](#), [98](#), [102](#), [119](#), [120](#), [122](#), [123](#), [124](#), [125](#), [126](#), [127](#), [128](#).
bracks: [28](#).
breakPending: [91](#), [109](#).
button: [91](#), [117](#).
buttonFile: [117](#).
c: [22](#), [52](#), [62](#), [63](#), [77](#), [103](#), [105](#), [129](#).
C_: [153](#).
C_LEFT_POINTING_DOUBLE_ANGLE_QUOTATION_MARK: [111](#), [131](#), [153](#).
C_RIGHT_POINTING_DOUBLE_ANGLE_QUOTATION_MARK: [111](#), [131](#), [139](#), [153](#).
c_str: [13](#), [15](#), [92](#), [100](#), [110](#), [117](#), [151](#).
centringTolerance: [52](#), [59](#).
cerr: [8](#), [29](#), [52](#), [69](#), [90](#), [91](#), [92](#), [101](#), [120](#), [140](#), [141](#), [148](#), [150](#), [151](#).
chap: [91](#), [100](#), [101](#), [115](#).
chapline: [91](#), [100](#), [101](#), [115](#).
chapname: [91](#), [98](#), [99](#), [100](#), [119](#), [126](#).
chapno: [91](#), [100](#), [101](#), [119](#), [126](#).
chapnumber: [91](#), [97](#), [99](#), [100](#), [119](#), [125](#), [126](#).
chapterCache: [91](#), [101](#).
ChapterMarker: [33](#), [36](#), [39](#), [40](#), [43](#), [48](#).
ChapterMarkerCharacter: [7](#), [43](#).
ChapterName: [40](#), [41](#), [48](#), [69](#), [92](#), [120](#).
ChapterNumber: [33](#), [36](#), [39](#), [48](#), [69](#), [92](#), [120](#).
check: [52](#), [53](#), [54](#), [55](#), [56](#), [57](#), [58](#), [59](#), [60](#), [61](#).
check_for: [52](#).
checkText: [141](#), [146](#), [148](#).
chtitle: [91](#), [100](#), [101](#).
cin: [13](#), [158](#).
classification: [42](#), [43](#).
classifyLine: [28](#), [37](#), [42](#), [48](#), [59](#), [158](#).
cleanText: [141](#), [146](#), [148](#).
close: [92](#), [101](#), [117](#).
cmd: [90](#), [119](#), [140](#).
compare: [24](#).
component: [141](#).
componentName: [8](#), [13](#), [15](#), [16](#), [18](#), [19](#), [21](#), [23](#), [24](#), [27](#), [28](#), [51](#), [52](#), [64](#), [68](#), [91](#), [119](#).
consecutive_blank_lines: [52](#), [60](#).
conversionTable: [23](#).
convert: [23](#).
convertForeignCharacterSetToISOFilter: [23](#), [141](#), [155](#), [158](#).
count: [91](#), [120](#).
cout: [15](#), [147](#), [148](#).
cp: [63](#), [77](#), [81](#), [82](#), [84](#), [85](#), [89](#), [103](#), [105](#), [107](#), [108](#), [111](#), [129](#), [131](#), [133](#), [134](#), [136](#), [137](#), [139](#), [141](#), [150](#).
cp850_to_ISO: [141](#), [155](#), [158](#).
createNavButton: [91](#), [116](#), [117](#), [158](#).
createNavButtons: [91](#), [100](#), [115](#), [116](#), [117](#), [158](#).
createNavigationPanel: [91](#), [101](#), [115](#), [158](#).
ct: [52](#).
ctime: [70](#), [91](#), [121](#).
currentOutput: [24](#), [25](#), [26](#).
d_foot: [116](#).
d_next: [116](#).
d_prev: [116](#).
d_up: [116](#).
debugParser: [143](#), [146](#), [148](#).
debugParserFile: [143](#), [146](#), [148](#).

- dec*: [58](#).
- decl*: [112](#).
- Declarations*: [31](#), [32](#), [48](#), [69](#), [92](#), [120](#).
- declarationsQueue*: [91](#), [93](#), [112](#), [158](#).
- DecodeBodyState*: [48](#), [64](#), [69](#), [92](#), [120](#).
- DefaultCentringTolerance*: [52](#).
- defaultFootnotePad*: [91](#).
- delim*: [90](#), [140](#).
- deque**: [66](#), [67](#), [91](#), [112](#).
- dest*: [9](#).
- destination*: [10](#), [13](#), [91](#), [119](#).
- disableAuditCriteria*: [52](#), [141](#), [158](#).
- DocumentTitle*: [31](#), [33](#), [48](#), [69](#), [92](#), [120](#).
- dosCharacters*: [141](#), [146](#), [148](#).
- dosconv*: [141](#).
- dostrip*: [13](#).
- dqueue*: [158](#).
- dubious_justification*: [52](#), [59](#).
- e*: [141](#).
- eflink*: [108](#).
- efn*: [92](#), [100](#).
- eh*: [63](#).
- elideNewLines*: [91](#), [100](#), [112](#), [158](#).
- em*: [53](#), [54](#), [55](#), [56](#), [57](#), [58](#), [59](#).
- embedded_tabs*: [52](#), [55](#), [58](#), [141](#), [158](#).
- emit*: [10](#), [12](#), [13](#), [18](#), [19](#), [21](#), [23](#), [24](#), [27](#), [28](#), [51](#), [52](#), [64](#), [68](#), [69](#), [70](#), [71](#), [72](#), [73](#), [74](#), [75](#), [76](#), [91](#), [102](#), [108](#), [109](#), [119](#), [120](#), [121](#), [122](#), [124](#), [126](#), [127](#), [128](#), [134](#), [135](#), [158](#).
- emitq*: [68](#), [72](#), [73](#), [74](#), [75](#), [91](#), [119](#).
- emitQueuedLines*: [28](#), [33](#), [39](#), [47](#).
- emits*: [28](#), [30](#), [31](#), [32](#), [33](#), [34](#), [35](#), [36](#), [37](#), [38](#), [40](#), [41](#), [47](#).
- empty*: [47](#), [101](#).
- enableAuditCriteria*: [52](#), [158](#).
- EncodeBodyState*: [28](#), [48](#).
- end*: [18](#), [20](#), [22](#), [23](#), [63](#), [67](#), [77](#), [82](#), [89](#), [91](#), [103](#), [105](#), [108](#), [111](#), [112](#), [120](#), [129](#), [131](#), [134](#), [139](#).
- End*: [32](#), [35](#), [37](#), [38](#), [41](#), [47](#), [49](#), [69](#), [72](#), [73](#), [74](#), [75](#), [76](#), [95](#), [97](#), [98](#), [102](#), [120](#), [124](#), [125](#), [126](#), [127](#), [128](#), [158](#).
- EndOfParagraph*: [48](#).
- EndOfText*: [28](#), [48](#), [69](#), [92](#), [101](#), [120](#), [158](#).
- envtype*: [68](#), [76](#), [119](#), [127](#), [128](#).
- eof*: [11](#), [12](#), [14](#), [25](#), [26](#), [27](#), [28](#).
- epilogueProcessor*: [24](#), [25](#), [26](#).
- epiP*: [24](#).
- erase*: [13](#), [18](#).
- err*: [52](#), [53](#), [54](#), [55](#), [56](#), [57](#), [58](#), [59](#), [60](#), [61](#).
- etextBodyParser*: [68](#), [91](#), [119](#).
- etextBodyParserFilter**: [28](#), [42](#), [44](#), [45](#), [46](#), [47](#), [51](#), [52](#), [59](#), [69](#), [71](#), [75](#), [76](#), [90](#), [92](#), [93](#), [102](#), [120](#), [122](#), [127](#), [128](#), [140](#), [141](#), [143](#), [158](#).
- everything*: [52](#), [158](#).
- exceeds_maximum_length*: [52](#), [57](#), [141](#).
- exit*: [29](#), [69](#), [91](#), [92](#), [120](#).
- f*: [28](#), [67](#), [141](#).
- false*: [13](#), [44](#), [51](#), [52](#), [59](#), [60](#), [62](#), [68](#), [69](#), [72](#), [75](#), [76](#), [90](#), [91](#), [94](#), [101](#), [108](#), [109](#), [119](#), [120](#), [123](#), [127](#), [128](#), [134](#), [135](#), [140](#), [146](#), [151](#), [158](#).
- fchar*: [43](#).
- filterType**: [8](#).
- FilterType*: [8](#), [9](#), [17](#).
- find*: [19](#), [44](#), [45](#), [46](#), [55](#), [56](#), [67](#), [90](#), [91](#), [140](#).
- find_first_not_of*: [42](#), [43](#), [44](#), [56](#), [59](#), [60](#), [63](#), [90](#), [91](#), [119](#), [140](#).
- find_first_of*: [42](#), [77](#), [89](#), [103](#), [111](#), [139](#).
- first*: [44](#), [45](#), [46](#).
- firstchap*: [68](#), [74](#), [91](#), [119](#).
- fiso*: [141](#).
- fitalics*: [91](#), [108](#), [109](#), [119](#), [134](#), [135](#).
- flattenISO*: [22](#), [156](#).
- flattenISOCharactersFilter**: [21](#), [141](#), [158](#).
- flattenISOchars*: [141](#), [146](#), [148](#).
- flink*: [134](#).
- flushBreak*: [91](#), [98](#), [108](#).
- foot*: [91](#), [92](#), [110](#).
- footdocname*: [91](#), [92](#), [108](#), [110](#).
- footline*: [91](#), [92](#), [110](#).
- footnest*: [68](#), [82](#), [83](#), [91](#), [108](#), [109](#), [119](#), [134](#), [135](#).
- FOOTNOTE_BUTTON_NEEDED**: [116](#).
- footnotePad*: [91](#), [109](#).
- footnotes*: [119](#), [120](#), [134](#), [135](#), [158](#).
- footnum*: [91](#), [108](#), [119](#), [120](#), [134](#), [135](#).
- footpar*: [119](#), [127](#), [128](#), [134](#), [135](#), [158](#).
- footsave*: [119](#), [134](#), [135](#), [158](#).
- FormatWidth*: [7](#), [42](#), [52](#), [141](#).
- fp*: [91](#).
- fquoth*: [119](#), [134](#), [135](#).
- frenchPunct*: [70](#), [77](#), [89](#), [111](#), [129](#), [131](#), [139](#), [146](#), [148](#), [158](#).
- from*: [23](#), [66](#).
- fromString*: [66](#), [67](#).
- front*: [47](#), [101](#).
- fType*: [8](#), [9](#), [12](#), [14](#), [17](#).
- fTypeName*: [8](#).
- generateAlignedParagraph*: [68](#), [69](#), [76](#), [91](#), [92](#), [102](#), [119](#), [120](#), [128](#), [129](#), [158](#).
- generateFilledParagraph*: [119](#), [120](#), [127](#), [129](#), [158](#).
- get*: [10](#), [12](#), [13](#).
- getAuditCriteria*: [52](#), [158](#).
- getBaseName*: [91](#).
- getCentringTolerance*: [52](#).
- getFootnotePad*: [91](#).

- getline*: [13](#).
- getLineNumber*: [8](#), [69](#), [120](#).
- getopt*: [145](#), [158](#).
- getopt_long*: [145](#), [148](#), [158](#).
- getSource*: [8](#), [9](#).
- getSourceLineNumber*: [8](#), [9](#), [141](#).
- getSpecialFilter*: [28](#).
- getStripEOL*: [13](#), [158](#).
- hasauthor*: [68](#), [73](#), [91](#), [95](#), [96](#), [112](#), [119](#), [124](#).
- hastitle*: [68](#), [72](#), [73](#), [91](#), [94](#), [96](#), [101](#), [112](#), [119](#), [123](#), [124](#).
- hauthor*: [91](#), [95](#), [96](#), [112](#), [119](#), [124](#).
- HAVE_STAT**: [145](#), [151](#).
- HAVE_UNISTD_H**: [145](#).
- heatSink**: [16](#), [141](#).
- hex*: [58](#), [63](#).
- hgs*: [141](#).
- hs*: [141](#).
- htitle*: [91](#), [94](#), [96](#), [100](#), [101](#), [110](#), [119](#), [123](#), [124](#).
- HTML**: [141](#), [146](#), [148](#), [150](#), [151](#).
- HTMLGenerationSink**: [91](#), [92](#), [102](#), [103](#), [105](#), [112](#), [113](#), [114](#), [115](#), [116](#), [117](#), [141](#), [158](#).
- i*: [13](#), [23](#), [42](#), [52](#), [67](#), [91](#), [141](#).
- Iabs*: [7](#), [59](#).
- identityTransform*: [23](#).
- ifstream**: [13](#).
- improper_embedded_blanks*: [52](#), [56](#).
- in*: [13](#).
- InBlockQuote*: [36](#), [37](#), [42](#), [48](#), [69](#), [92](#), [120](#).
- InCentred*: [31](#), [33](#), [34](#), [35](#), [36](#), [39](#), [40](#), [41](#), [43](#), [48](#), [59](#), [69](#), [92](#), [120](#).
- index*: [91](#), [92](#), [95](#), [96](#), [98](#), [99](#), [100](#).
- indexFileName*: [91](#), [92](#), [101](#).
- indexline*: [91](#), [92](#), [96](#), [99](#), [100](#).
- infile*: [141](#), [149](#), [150](#), [151](#).
- infoot*: [91](#), [102](#), [108](#), [109](#), [119](#), [127](#), [128](#), [134](#), [135](#), [158](#).
- inmath*: [68](#), [77](#), [81](#), [89](#), [91](#), [105](#), [107](#), [111](#), [119](#), [129](#), [131](#), [133](#), [139](#).
- inParagraph*: [91](#), [115](#).
- InPreformattedTable*: [36](#), [38](#), [42](#), [48](#), [59](#), [69](#), [92](#), [120](#), [128](#).
- InRaggedLeft*: [36](#), [37](#), [42](#), [48](#), [69](#), [92](#), [120](#).
- InRaggedRight*: [36](#), [37](#), [42](#), [48](#), [69](#), [92](#), [120](#).
- insource*: [141](#).
- instat*: [151](#).
- inTable*: [52](#), [56](#), [59](#).
- intable*: [68](#), [69](#), [77](#), [119](#), [120](#), [129](#), [139](#).
- interval*: [19](#).
- InTextParagraph*: [36](#), [37](#), [42](#), [48](#), [69](#), [92](#), [120](#).
- invalid_argument**: [10](#), [12](#), [13](#), [14](#), [141](#), [158](#).
- invalid_characters*: [52](#), [58](#), [141](#).
- io_dup*: [151](#).
- ios*: [13](#), [15](#), [92](#), [100](#), [110](#), [117](#).
- is*: [13](#).
- isCharacterPermissible*: [52](#), [58](#), [62](#), [63](#), [158](#).
- isISOletter*: [52](#), [54](#), [158](#).
- isLineSpecial*: [28](#), [31](#), [32](#), [44](#), [45](#), [51](#), [52](#), [71](#), [75](#), [76](#), [93](#), [102](#), [122](#), [127](#), [128](#).
- isochar*: [131](#).
- isspace*: [18](#), [53](#), [78](#), [89](#), [104](#), [111](#), [130](#), [131](#), [158](#).
- isSub*: [90](#), [140](#).
- isSubstitution*: [68](#), [71](#), [75](#), [76](#), [90](#), [119](#), [122](#), [127](#), [128](#), [140](#).
- issueMessage*: [8](#), [52](#), [53](#), [54](#), [55](#), [56](#), [57](#), [58](#), [59](#), [60](#), [61](#), [83](#), [90](#), [108](#), [109](#), [134](#), [135](#), [140](#).
- istream**: [13](#).
- italics*: [68](#), [80](#), [91](#), [106](#), [108](#), [109](#), [119](#), [132](#), [134](#), [135](#), [158](#).
- iterator**: [20](#), [22](#), [23](#), [63](#), [67](#), [77](#), [103](#), [105](#), [112](#), [129](#), [158](#).
- j*: [53](#).
- l*: [59](#), [90](#), [109](#), [128](#), [140](#).
- last*: [45](#), [46](#).
- lastBlank*: [51](#), [52](#), [60](#).
- lastStripped*: [51](#).
- LaTeX*: [141](#), [146](#), [148](#).
- LaTeXGenerationFilter**: [68](#), [69](#), [76](#), [77](#), [90](#), [141](#), [158](#).
- lchar*: [43](#).
- lclass*: [52](#), [59](#).
- length*: [13](#), [18](#), [42](#), [43](#), [44](#), [51](#), [53](#), [54](#), [55](#), [57](#), [58](#), [59](#), [64](#), [67](#), [70](#), [91](#), [117](#), [121](#), [139](#).
- lf*: [141](#).
- lineClass*: [28](#), [29](#), [31](#), [33](#), [34](#), [35](#), [36](#), [37](#), [38](#), [39](#), [40](#), [41](#).
- lineCounter*: [91](#), [112](#), [113](#), [114](#).
- lineNumber*: [8](#), [10](#), [14](#), [91](#).
- linesIn*: [91](#), [99](#), [100](#), [101](#), [158](#).
- log*: [52](#), [53](#), [54](#), [55](#), [56](#), [57](#), [58](#), [59](#), [60](#), [61](#).
- long_options*: [148](#).
- lq*: [28](#), [31](#), [33](#), [36](#), [39](#), [47](#).
- m*: [90](#), [140](#).
- main*: [141](#).
- make_one_file*: [91](#).
- MarkerMinimumLength*: [7](#), [43](#).
- mathModeQuoted*: [77](#).
- maxlen*: [52](#).
- maxLineLength*: [52](#), [57](#), [59](#), [158](#).
- msg*: [8](#).
- n*: [20](#), [52](#), [67](#), [90](#), [140](#).
- NETSCRAPE_SUCKS**: [108](#), [109](#).
- next*: [91](#), [115](#).

- npos*: [19](#), [42](#), [43](#), [44](#), [45](#), [46](#), [55](#), [56](#), [60](#), [63](#), [67](#), [77](#), [89](#), [90](#), [91](#), [111](#), [139](#), [140](#).
- nsep*: [24](#), [25](#), [26](#).
- numchap*: [126](#).
- o*: [15](#), [45](#), [46](#), [63](#), [67](#), [77](#), [91](#), [103](#), [105](#), [129](#).
- of*: [8](#).
- ofilt*: [9](#), [14](#).
- ofmt*: [141](#), [146](#), [148](#), [150](#), [151](#).
- ofstream**: [15](#), [91](#), [92](#), [100](#), [110](#), [117](#).
- openFile*: [13](#), [141](#), [158](#).
- opt*: [141](#), [148](#).
- optarg*: [148](#).
- optind*: [150](#).
- option*: [148](#).
- option_index*: [148](#).
- os*: [15](#), [20](#), [22](#), [52](#), [91](#), [112](#), [113](#), [114](#), [141](#).
- ostream**: [8](#), [15](#), [52](#), [91](#), [112](#), [113](#), [114](#).
- ostreamstream**: [53](#), [54](#), [55](#), [56](#), [57](#), [58](#), [59](#), [63](#), [92](#), [108](#), [126](#), [131](#), [134](#), [135](#).
- out*: [15](#), [92](#), [100](#), [110](#), [117](#).
- outfile*: [141](#), [149](#), [150](#), [151](#).
- output*: [8](#), [9](#), [10](#), [11](#), [17](#), [24](#), [25](#), [26](#).
- outputFormat**: [146](#).
- outstat*: [151](#).
- p*: [20](#), [22](#), [23](#), [54](#).
- Palm*: [141](#), [146](#), [148](#).
- PalmGenerationFilter**: [119](#), [120](#), [127](#), [128](#), [129](#), [140](#), [141](#), [158](#).
- parline*: [119](#), [128](#).
- parserDiagnosticFilter**: [50](#), [64](#), [143](#).
- partext*: [119](#), [127](#).
- pathName*: [13](#), [15](#).
- pd*: [143](#).
- pdsink*: [143](#).
- pdtsq*: [143](#).
- permit_8_bit_ISO_characters*: [52](#), [58](#), [141](#), [158](#).
- pf*: [141](#).
- pipeEnd*: [141](#), [158](#).
- Plumb*: [141](#), [143](#), [158](#).
- pop*: [47](#), [101](#).
- PossibleChapterNumber*: [36](#), [39](#), [48](#).
- PossibleTitle*: [31](#), [33](#), [48](#).
- postambleLines*: [114](#).
- preambleLines*: [112](#).
- PreformattedTableIndent*: [7](#), [38](#), [42](#).
- prev*: [91](#), [115](#).
- prodest*: [142](#).
- PRODUCT**: [1](#), [70](#), [112](#), [121](#), [147](#), [148](#).
- prologueProcessor*: [24](#).
- proP*: [24](#).
- pruneIndent*: [91](#), [94](#), [95](#), [97](#), [98](#), [119](#), [123](#), [124](#), [125](#), [126](#), [127](#), [128](#), [158](#).
- punctuation*: [77](#), [89](#), [105](#), [111](#), [129](#), [139](#).
- PUNCTUATION**: [7](#), [105](#).
- push*: [31](#), [33](#), [36](#), [39](#), [91](#).
- push_back*: [66](#), [93](#).
- put*: [8](#), [10](#), [12](#), [14](#), [15](#), [16](#), [17](#), [18](#), [19](#), [21](#), [23](#), [24](#), [27](#), [28](#), [29](#), [51](#), [52](#), [64](#), [68](#), [69](#), [91](#), [92](#), [119](#), [120](#).
- qchapname*: [91](#), [100](#), [101](#).
- qchapnumber*: [91](#), [100](#), [101](#).
- queue**: [28](#), [91](#).
- quoteArbitraryString*: [52](#), [63](#), [90](#), [140](#), [158](#).
- quotedCharacters*: [77](#), [89](#).
- quotedTextCharacters*: [77](#), [89](#).
- quoteHTMLString*: [91](#), [94](#), [95](#), [97](#), [98](#), [103](#), [105](#).
- QuoteIndent*: [7](#), [42](#), [43](#).
- quoteLaTeXString*: [68](#), [69](#), [76](#), [77](#), [158](#).
- quotePalmString*: [119](#), [120](#), [123](#), [124](#), [125](#), [126](#), [127](#), [128](#), [129](#), [135](#), [158](#).
- quoth*: [68](#), [75](#), [76](#), [88](#), [119](#), [127](#), [128](#), [134](#), [135](#), [138](#), [158](#).
- r*: [59](#).
- RaggedRightIndent*: [7](#), [42](#), [43](#).
- REGISTERED_SIGN**: [147](#), [153](#).
- replace*: [67](#), [91](#).
- REVDATE**: [1](#), [70](#), [112](#), [121](#), [148](#).
- rfind*: [44](#), [46](#).
- RIGHT_POINTING_DOUBLE_ANGLE_QUOTATION_MARK**: [7](#), [153](#).
- rtype*: [64](#).
- s*: [8](#), [10](#), [12](#), [13](#), [14](#), [15](#), [16](#), [18](#), [19](#), [21](#), [23](#), [24](#), [27](#), [28](#), [42](#), [44](#), [45](#), [46](#), [51](#), [52](#), [63](#), [64](#), [66](#), [67](#), [68](#), [69](#), [77](#), [90](#), [91](#), [92](#), [101](#), [102](#), [103](#), [105](#), [119](#), [120](#), [126](#), [127](#), [128](#), [129](#), [140](#).
- saveEpilogueFile*: [142](#), [146](#), [148](#).
- savePrologueFile*: [142](#), [146](#), [148](#).
- secondDestination*: [27](#).
- secP*: [27](#).
- sectionSep*: [24](#), [25](#).
- sectionSeparatorSquid**: [24](#), [26](#), [28](#), [64](#), [141](#), [142](#).
- send*: [12](#), [141](#).
- sentenceEnd*: [52](#), [56](#).
- setAuditCriteria*: [52](#), [141](#), [158](#).
- setCentringTolerance*: [52](#).
- setConversionTable*: [23](#).
- setEpilogueProcessor*: [24](#), [142](#).
- setfill*: [63](#), [131](#).
- setFootnotePad*: [91](#).
- setLogStream*: [52](#).
- setMaxLength*: [52](#).
- setOutput*: [9](#), [14](#).
- setPrologueProcessor*: [24](#), [142](#).
- setSpecialFilter*: [28](#), [141](#), [158](#).
- setStripEOL*: [13](#), [141](#), [158](#).

- setTabInterval*: [19](#).
- setTranslation*: [23](#).
- setw*: [63](#), [131](#).
- singleFile*: [91](#), [92](#), [97](#), [98](#), [108](#), [109](#).
- singleFileHTML*: [141](#), [146](#), [148](#).
- SinkType*: [8](#), [9](#), [14](#).
- source*: [8](#), [9](#), [12](#), [91](#), [117](#).
- SourceType*: [8](#), [9](#), [12](#).
- spaces*: [64](#).
- special*: [52](#), [56](#), [57](#), [59](#), [61](#).
- special_commands_present*: [52](#), [61](#), [141](#).
- specialCommand*: [28](#), [46](#), [71](#), [75](#), [76](#), [90](#), [93](#), [102](#), [122](#), [127](#), [128](#), [140](#).
- specialFilter*: [28](#).
- SpecialMarker*: [7](#), [44](#), [46](#).
- SpecialPrefix*: [7](#), [44](#), [45](#), [46](#).
- specialStrip*: [141](#), [146](#), [148](#).
- specialType*: [28](#), [45](#).
- squiddley*: [141](#), [142](#).
- ssc*: [141](#).
- st_dev*: [151](#).
- st_ino*: [151](#).
- stat*: [151](#), [158](#).
- state*: [28](#), [29](#), [30](#), [31](#), [32](#), [33](#), [34](#), [35](#), [36](#), [37](#), [38](#), [39](#), [40](#), [41](#), [69](#), [92](#), [101](#), [119](#), [120](#), [127](#), [128](#).
- stateName*: [64](#).
- stateNames*: [29](#), [50](#), [64](#), [69](#), [92](#), [120](#).
- std**: [145](#).
- stime*: [70](#), [91](#), [112](#), [121](#).
- str*: [53](#), [54](#), [55](#), [56](#), [57](#), [58](#), [59](#), [63](#), [100](#), [108](#), [126](#), [131](#), [134](#), [135](#).
- streamSink**: [15](#), [16](#), [141](#), [142](#), [143](#), [158](#).
- streamSource**: [13](#), [141](#), [158](#).
- string**: [8](#), [10](#), [12](#), [13](#), [14](#), [15](#), [16](#), [18](#), [19](#), [20](#), [21](#), [22](#), [23](#), [24](#), [27](#), [28](#), [42](#), [43](#), [44](#), [45](#), [46](#), [50](#), [51](#), [52](#), [55](#), [56](#), [60](#), [63](#), [64](#), [66](#), [67](#), [68](#), [69](#), [70](#), [76](#), [77](#), [89](#), [90](#), [91](#), [92](#), [101](#), [102](#), [103](#), [105](#), [111](#), [112](#), [113](#), [117](#), [119](#), [120](#), [121](#), [126](#), [127](#), [128](#), [129](#), [139](#), [140](#), [146](#), [149](#).
- strip*: [13](#).
- stripSpecialCommandsFilter**: [51](#), [61](#), [141](#), [158](#).
- substitute*: [66](#), [67](#), [77](#), [129](#).
- substr*: [45](#), [46](#), [56](#), [64](#), [69](#), [70](#), [90](#), [91](#), [92](#), [119](#), [120](#), [121](#), [128](#), [140](#).
- t*: [67](#), [70](#), [91](#), [105](#), [121](#).
- tabExpanderFilter**: [19](#), [62](#), [141](#).
- tabf*: [141](#).
- tabInterval*: [19](#), [20](#).
- tbl*: [23](#).
- teeSquid**: [27](#), [143](#).
- terminator*: [68](#), [76](#), [91](#), [102](#), [158](#).
- texform*: [79](#), [154](#).
- text*: [28](#), [68](#), [69](#), [71](#), [72](#), [73](#), [74](#), [75](#), [76](#), [91](#), [92](#), [93](#), [94](#), [95](#), [97](#), [98](#), [102](#), [119](#), [120](#), [122](#), [123](#), [124](#), [125](#), [126](#), [127](#), [128](#).
- textComponent**: [8](#), [9](#), [10](#), [12](#), [13](#), [14](#), [17](#), [24](#), [27](#), [52](#), [91](#), [119](#), [141](#), [142](#).
- textFilter**: [17](#), [18](#), [19](#), [21](#), [23](#), [24](#), [27](#), [28](#), [51](#), [52](#), [64](#), [68](#), [119](#).
- textSink**: [14](#), [15](#), [16](#), [91](#).
- textSource**: [10](#), [12](#), [13](#).
- textSubstituter**: [66](#), [67](#), [68](#), [119](#).
- tfilt*: [141](#).
- time*: [70](#), [91](#), [121](#).
- title*: [91](#), [112](#), [113](#).
- TitleMarker*: [31](#), [33](#), [34](#), [36](#), [43](#), [48](#).
- TitleMarkerCharacter*: [7](#), [43](#).
- to*: [23](#), [66](#).
- toString*: [66](#), [67](#).
- trailing-blanks*: [52](#), [53](#), [141](#).
- trailing-hyphen*: [52](#), [54](#), [158](#).
- transformer*: [68](#), [77](#), [90](#), [119](#), [129](#), [140](#).
- translateHTMLString*: [91](#), [92](#), [96](#), [99](#), [100](#), [101](#), [102](#), [105](#), [158](#).
- trimFilter**: [18](#), [62](#), [141](#).
- true*: [29](#), [44](#), [45](#), [51](#), [53](#), [54](#), [55](#), [56](#), [57](#), [58](#), [59](#), [60](#), [61](#), [62](#), [69](#), [72](#), [73](#), [74](#), [90](#), [91](#), [94](#), [95](#), [108](#), [109](#), [120](#), [123](#), [124](#), [134](#), [140](#), [141](#), [148](#), [151](#), [158](#).
- UndefinedType*: [8](#).
- usage*: [147](#), [148](#), [158](#).
- verbose*: [69](#), [92](#), [101](#), [120](#), [141](#), [146](#), [148](#).
- VERSION**: [1](#), [70](#), [112](#), [121](#), [148](#).
- Void*: [28](#), [30](#), [31](#), [33](#), [34](#), [36](#), [40](#), [49](#), [72](#), [73](#), [74](#), [75](#), [76](#), [94](#), [95](#), [97](#), [98](#), [102](#), [123](#), [124](#), [125](#), [126](#), [127](#), [128](#).
- what*: [141](#).
- write*: [117](#).
- writeDescription*: [8](#).
- writeHTMLDocumentBodyStart*: [91](#), [96](#), [101](#), [110](#), [113](#).
- writeHTMLDocumentPostamble*: [91](#), [92](#), [101](#), [114](#).
- writeHTMLDocumentPreamble*: [91](#), [96](#), [101](#), [110](#), [112](#), [158](#).

- ⟨ Author state 35 ⟩ Used in section 29.
- ⟨ BeforeTitle state 31 ⟩ Used in section 29.
- ⟨ Begin footnote in HTML 108 ⟩ Used in section 105.
- ⟨ Begin footnote in LaTeX 82 ⟩ Used in section 77.
- ⟨ Begin footnote in Palm 134 ⟩ Used in section 129.
- ⟨ BeginText state 30 ⟩ Used in section 29.
- ⟨ BetweenParagraphs state 36 ⟩ Used in section 29.
- ⟨ ChapterMarker state 40 ⟩ Used in section 29.
- ⟨ ChapterName state 41 ⟩ Used in section 29.
- ⟨ Check for consecutive blank lines 60 ⟩ Used in section 52.
- ⟨ Check for input and output files the same 151 ⟩ Used in section 150.
- ⟨ Check for invalid characters in text 58 ⟩ Cited in section 55. Used in section 52.
- ⟨ Check for justification-related problems 59 ⟩ Used in section 52.
- ⟨ Check for line that exceeds maximum text length 57 ⟩ Used in section 52.
- ⟨ Check for line with embedded tab characters 55 ⟩ Used in section 52.
- ⟨ Check for line with improper embedded white space 56 ⟩ Used in section 52.
- ⟨ Check for line with trailing hyphen 54 ⟩ Used in section 52.
- ⟨ Check for line with trailing white space 53 ⟩ Cited in section 158. Used in section 52.
- ⟨ Check for special commands present 61 ⟩ Used in section 52.
- ⟨ Class definitions 8, 12, 13, 14, 15, 16, 17, 18, 19, 21, 23, 24, 27, 28, 42, 44, 45, 46, 47, 51, 52, 62, 63, 64, 66, 67, 68, 69, 76, 77, 90, 91, 92, 102, 103, 105, 112, 113, 114, 115, 116, 117, 119, 120, 127, 128, 129, 140 ⟩ Used in section 6.
- ⟨ Classify centred line 43 ⟩ Used in section 42.
- ⟨ Complete chapter file generation in HTML 101 ⟩ Used in sections 92 and 100.
- ⟨ Configure prologue and epilogue processing 142 ⟩ Used in section 141.
- ⟨ Connect components in pipeline 9 ⟩ Used in section 8.
- ⟨ Convert ASCII quotes to open and close quotes in LaTeX 88 ⟩ Used in section 77.
- ⟨ Convert ASCII quotes to open and close quotes in Palm 138 ⟩ Used in section 129.
- ⟨ Create footnote file for first footnote in HTML 110 ⟩ Used in section 108.
- ⟨ Declarations state 32 ⟩ Used in section 29.
- ⟨ Definition of navigation buttons in HTML 118 ⟩ Used in section 116.
- ⟨ Emit output to next component in pipeline 10 ⟩ Used in section 8.
- ⟨ End footnote in HTML 109 ⟩ Used in section 105.
- ⟨ End footnote in LaTeX 83 ⟩ Used in section 77.
- ⟨ End footnote in Palm 135 ⟩ Used in section 129.
- ⟨ Expand tabs in text line 20 ⟩ Used in section 19.
- ⟨ Flatten ISO 8859 characters to 7-bit ASCII 22 ⟩ Used in section 21.
- ⟨ Generate chapter title for document tree output in HTML 100 ⟩ Used in section 98.
- ⟨ Generate chapter title for single file output in HTML 99 ⟩ Used in section 98.
- ⟨ Generate index document header in HTML 96 ⟩ Used in section 95.
- ⟨ Generate justified text paragraph in LaTeX 75 ⟩ Used in section 69.
- ⟨ Generate start of document in LaTeX 70 ⟩ Used in section 69.
- ⟨ Generate start of document in Palm 121 ⟩ Used in section 120.
- ⟨ Global functions 147 ⟩ Used in section 6.
- ⟨ Global variables 48, 49, 50, 146, 149, 154, 155, 156 ⟩ Used in section 6.
- ⟨ Handle end of file notification 11 ⟩ Used in section 8.
- ⟨ Handle section separator 25 ⟩ Used in section 24.
- ⟨ Main program 141 ⟩ Used in section 6.
- ⟨ Output ASCII text character in LaTeX 89 ⟩ Used in section 77.
- ⟨ Output ASCII text character in Palm 139 ⟩ Cited in section 158. Used in section 129.
- ⟨ Output text character in HTML 111 ⟩ Used in section 105.
- ⟨ Parse command-line file arguments 150 ⟩ Used in section 141.
- ⟨ Parser state machine 29 ⟩ Used in section 28.

⟨PossibleChapterNumber state 39⟩ Used in section 29.
 ⟨PossibleTitle state 33⟩ Used in section 29.
 ⟨Process author in HTML 95⟩ Used in section 92.
 ⟨Process author in LaTeX 73⟩ Used in section 69.
 ⟨Process author in Palm 124⟩ Used in section 120.
 ⟨Process chapter name in HTML 98⟩ Used in section 92.
 ⟨Process chapter name in LaTeX 74⟩ Used in section 69.
 ⟨Process chapter name in Palm 126⟩ Used in section 120.
 ⟨Process chapter number in HTML 97⟩ Used in section 92.
 ⟨Process chapter number in Palm 125⟩ Used in section 120.
 ⟨Process command-line options 148⟩ Used in section 141.
 ⟨Process declarations in HTML 93⟩ Used in section 92.
 ⟨Process declarations in LaTeX 71⟩ Used in section 69.
 ⟨Process declarations in Palm 122⟩ Used in section 120.
 ⟨Process document title in HTML 94⟩ Used in section 92.
 ⟨Process document title in LaTeX 72⟩ Used in section 69.
 ⟨Process document title in Palm 123⟩ Used in section 120.
 ⟨Program implementation 6⟩ Used in section 5.
 ⟨Quote ASCII character as verbatim in LaTeX 86⟩ Used in section 77.
 ⟨Quote ISO 8859-1 character in Palm 131⟩ Cited in section 158. Used in section 129.
 ⟨Quote character as math mode in LaTeX 87⟩ Used in section 77.
 ⟨Quote control character in HTML 104⟩ Used in section 103.
 ⟨Quote control character in LaTeX 78⟩ Used in section 77.
 ⟨Quote control character in Palm 130⟩ Used in section 129.
 ⟨Section separator squid end of file handling 26⟩ Used in section 24.
 ⟨Set up parser debugging if requested 143⟩ Used in section 141.
 ⟨System include files 145⟩ Used in section 5.
 ⟨TitleMarker state 34⟩ Used in section 29.
 ⟨Toggle italic text mode in HTML 106⟩ Used in section 105.
 ⟨Toggle italic text mode in LaTeX 80⟩ Used in section 77.
 ⟨Toggle italic text mode in Palm 132⟩ Used in section 129.
 ⟨Toggle math mode in HTML 107⟩ Used in section 105.
 ⟨Toggle math mode in LaTeX 81⟩ Used in section 77.
 ⟨Toggle math mode in Palm 133⟩ Used in section 129.
 ⟨Translate ISO graphic character in LaTeX 79⟩ Used in section 77.
 ⟨Translate ellipsis in LaTeX 85⟩ Used in section 77.
 ⟨Translate ellipsis in Palm 137⟩ Used in section 129.
 ⟨Translate em-dash in LaTeX 84⟩ Used in section 77.
 ⟨Translate em-dash in Palm 136⟩ Used in section 129.
 ⟨Within aligned paragraph state 37⟩ Used in section 29.
 ⟨Within preformatted table state 38⟩ Used in section 29.

ETSET

	Section	Page
Introduction	1	1
Command line	2	2
Options	3	3
Input format	4	5
Program global context	5	8
Text processing components	8	9
Source components	12	12
Stream source	13	13
Sink components	14	15
Stream sink	15	16
Heat sink	16	17
Filter components	17	18
Trim filter	18	18
Tab expander filter	19	19
Flatten ISO characters filter	21	20
Convert foreign character set to ISO filter	23	21
Section separator squid	24	22
Tee squid	27	25
Etext body parser filter	28	26
Strip special commands filter	51	37
Audit filter	52	38
Parser diagnostic filter	64	45
Utilities	65	46
Text substituter	66	46
LaTeX Generation	68	47
HTML Generation	91	58
Palm Markup Language Generation	119	79
Main program	141	94
Application plumbing	144	97
Character set definitions and translation tables	152	104
ISO 8859-1 special characters	153	105
LaTeX representation of ISO graphic characters	154	106
MS-DOS code page 850 to ISO translation table	155	107
Flat 7-bit ASCII approximation of ISO characters	156	108
Release history	157	110
Development log	158	111
Index	159	123