1. Introduction.

The Annoyance Filter

by John Walker

Business propaganda must be obtrusive and blatant. It is its aim to attract the attention of slow people, to rouse latent wishes, to entice men to substitute innovation for inert clinging to traditional routine. In order to succeed, advertising must be adjusted to the mentality of the people courted. It must suit their tastes and speak their idiom. Advertising is shrill, noisy, coarse, puffing, because the public does not react to dignified allusions. It is the bad taste of the public that forces the advertisers to display bad taste in their publicity campaigns.

—Ludwig von Mises, *Human Action*

This program implements an adaptive Bayesian filter which distinguishes junk mail from legitimate mail by scanning archives of each and calculating the probability for each word which appears a statistically significant number of times in the body of text that the word will appear in junk mail.

After building a database of word probabilities, arriving mail is parsed into a list of unique words which are looked up in the probability database. A short list of words with extremal probability (most likely to identify a message as legitimate or as junk) is used to compute an aggregate message probability with Bayes' theorem. This probability is then tested against a threshold to decide whether the message as a whole is junk. Mail determined to be junk or legitimate can be added to the database to refine the probability values and adapt as the content of mail evolves over time. Ideally, this could be triggered to a button in a mail reader which dispatched a message to the appropriate category.

The technique and algorithms used by this program are as described in Paul Graham's "*A Plan for Spam*[1]". This C++ program was developed based on the model Common Lisp code in his document which, in turn, was modeled on the original code in the "Arc" language he is developing.

The concept of an adaptive advertising filter and the name of this program first appeared in my 1989 science fiction story "*We'll Return, After This Message*".

A complete development log giving the detailed history of this program appears at the end of this document.

```
#define REVDATE "2002-10-22"
#define Xfile string("X-Annoyance-Filter")
```

[1] SPAM® is a registered trademark of Hormel Foods Corporation. Use of the word to denote unsolicited commercial E-mail is based on the Monty Python skit in which a bunch of Vikings sing a chorus of "SPAM, SPAM, SPAM," drowning out all civil discourse. To avoid confusion with processed meat products, I use the term "junk mail" in this document. Besides, if "spam" is strictly defined as unsolicited commercial E-mail, the mandate of this program covers the much broader spectrum of *undesired* mail regardless of provenance and motivation.

2. User Guide.

`annoyance-filter` is invoked with a command line as follows:

> `annoyance-filter` *options*

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.

3. Getting started.

The Annoyance Filter is organised as a toolbox which can be used to explore content-based mail filtering. It includes diagnostic tools and output which will eventually be little used once the program is tuned and put into production.

The program is normally run in two phases. In the *training* phase, collections of legitimate and junk mail stored in UNIX mail folders are read and used to build a dictionary in which the probability of a word's identifying a message as junk is computed. This dictionary is then exported to be used in subsequent runs to classify incoming messages based on the word probabilities determined from prior messages.

3.1. Building

If you have a more or less standard present-day UNIX system, you should be able to build and install the program with the commands:

```
./configure
make
make check
make install
```

3.2. Training

Now you must *train* the program to discriminate legitimate junk and mail by showing it collections of such mail you've hand sorted into a pile of stuff you want to receive and another which you don't. Assuming you have mail folders containing collections of legitimate mail and junk named "`m-good`" and "`m-junk`" respectively, you can perform the training phase and create a binary dictionary file named "`dict.bin`" with the command:

```
annoyance-filter --mail m-good --junk m-junk --prune --write dict.bin
```

The arguments to the `--mail` and `--junk` options can be either UNIX "mail folders" consisting of one or more E-mail messages concatenated into a single file, or the name of a directory containing messages in individual files. In either case, the files may be compressed with `gzip`—`annoyance-filter` will automatically expand them. You can supply as many `--mail` and `--junk` options as you like on a command line; the contents added cumulatively to the dictionary.

It is *absolutely essential* that the collections of legitimate and junk mail used to train `annoyance-filter` be completely clean—no junk in the `--mail` collection or vice versa. Pollution of either collection by messages belonging in the other is very likely to corrupt the calculation of probabilities, resulting in messages which belong in one category being assigned to the other. The `utilities/splitmail.pl` program can help in manually sorting mail into the required two piles, and I hope some day I will have the time to adequately document it.

You may find it worthwhile to add an archive of mail you've sent to the legitimate category with `--mail`. In many cases, the words you use in mail you send are an excellent predictor of how worthy an incoming message is of your attention. I've found this works well with my own archives, but I haven't tested how effective it is for a broader spectrum of users.

When you compile the collections of junk and legitimate mail to train `annoyance-filter`, it's important to include *all* the copies of similar or identical messages you've received in either category. `annoyance-filter` bases its classifications on the frequency of indicative words in the entire set of mail you receive. An obscure string embedded in a mail worm spewed onto the net may not filter it out if you train `annoyance-filter` with only one copy, but will certainly consign it to the junk heap if you train `annoyance-filter` with the twenty or thirty you receive a day.

3.3. Scoring

Dictionary in hand, you can now proceed to the `scoring` phase, where the dictionary is used, along with the list of words appearing in a message, to determine its overall probability of being junk. If you have a mail message in a file "`mail.txt`", you can compute and display its junk probability with:

```
annoyance-filter --read dict.bin --test mail.txt
```

The probability is written to standard output. The closer the probability is to 1, the more likely the mail is junk.

3.4. Plumbing

To use annoyance-filter as a front-end to another mail filtering program, specify the --transcript option before --test—the junk probability and classification will be appended to the message header and written to the designated transcript destination, standard output if "-". For example, to use annoyance-filter as a front-end to a mail sorting program such as Procmail, you might invoke it with the command:

```
annoyance-filter --read dict.bin --transcript - --test -
```

which reads the message to be classified from standard input and writes the transcript, classification included, to standard output. Note that since the command line options are processed as commands, not stateless mode specifications, you must request the --transcript before designating the message to --test.

3.5. Progressive Refinement

Junk mail evolves, but annoyance-filter evolves *with it.* As incoming mail arrives and annoyance-filter sorts it into legitimate and junk categories, there will doubtless be the occasional error. The classification defaults used by annoyance-filter have been chosed that the vast majority of such error are in the direction of considering junk mail legitimate as opposed to the opposite, whose consequences are much more serious.

As annoyance-filter sorts your incoming mail, you'll amass folders of junk and non-junk it's classified, including the occasional error. If you take the time to go through these folders and sort out the occasional mis-classified messages, then add them to the annoyance-filter dictionary, the precision with which it classifies incoming messages will be increasingly refined. For example, suppose your current dictionary is dict.bin and you have sorted out folders of legitimate mail new-good and junk new-junk which have arrived since you built the dictionaty. You can update the dictionary based on new messages with the command:

```
annoyance-filter --read dict.bin --mail new-good --junk new-junk \
                 --prune --write dict.bin
```

Perhaps some day a mail client will provide a "Delete as junk" button which automatically discards the offending message and forwards it to annoyance-filter to further refine its criteria for identifying junk.

4. Options.

Options are specified on the command line. Options are treated as commands—most instruct the program to perform some specific action; consequently, the order in which they are specified is significant; they are processed left to right. Long options beginning with "`--`" may be abbreviated to any unambiguous prefix; single-letter options introduced by a single "`-`" without arguments may be aggregated.

`--annotate` *options*
: Add the annotations requested by the characters in *options* to the transcript generated by the `--transcript` option. Upper and lower case *options* are treated identically. Available annotations are:

`d`	Decoder diagnostics
`p`	Parser warnings and error messages
`w`	Most significant words and their probabilities

`--binword` *n*
: Binary character streams (for example, attachments of application-specific files, including the executable code of worm and virus attachments) are scanned and contiguous sequences of alphanumeric ASCII characters *n* characters or longer are added to the list of words in the message. The dollar sign ("`$`") is considered an alphanumeric character for these purposes, and words may have embedded hyphens and apostrophes, but may not begin or end with those characters. If `--binword` is set to zero, scanning of binary attachments is disabled entirely. The default setting is 5 characters.

`--classify` *fname*
: Classify mail in *fname*. If it equals or exceeds the junk threshold (see `--threshjunk`), "`JUNK`" is written to standard output and the program exits with status code 3. If the message scores less than or equal to the mail threshold (see `--threshmail`), "`MAIL`" is written to standard output and the program exits with status 0. If the message's score falls between the two thresholds, its content is deemed indeterminate; "`INDT`" is written to standard output and the program exits with a status of 4. The output can be used to set an environment variable in `Procmail` to control the disposition of the message. If *fname* is "`-`" the message is read from standard input.

`--clearjunk`
: Clear appearances of words in junk mail from database. Used when preparing a database of legitimate mail.

`--clearmail`
: Clear appearances of words in legitimate mail from database. Used when preparing a database of junk mail.

`--copyright`
: Print copyright information.

`--csvread` *fname*
: Import a dictionary from a comma-separated value (CSV) file *fname*. Records are assumed to be in the format written by `--csvwrite` but need not be sorted in any particular order. Words are added to those already in memory.

`--csvwrite` *fname*
: Export a dictionary as a comma-separated value (CSV) *fname* with this option. Such files can be loaded into spreadsheet or database programs for further processing. Words are sorted first in ascending order of probability they denote junk mail, then lexically.

`--help`, `-u`
: Print how-to-call information including a list of options.

`--junk`, `-j` *fname*
: Add the mail in folder *fname* to the dictionary as junk mail. These folders may be compressed by a utility the host system can uncompress; specify the complete file name including the extension denoting its form of compression. If *fname* is "`-`" the mail folder is read from standard input.

`--list`
: List the dictionary on standard output.

`--mail`, `-m` *fname*
: Add the mail in folder *fname* to the dictionary as legitimate mail. These folders may be compressed by a utility the host system can uncompress; specify the complete file name including the extension denoting its form of compression. If *fname* is "`-`" the mail folder is read from standard input.

`--newword` *n*
: The probability that a word seen in mail which does not appear in the dictionary (or appeared too few times to assign it a probability with acceptable confidence) is indicative of junk is set to *n*. The default is 0.2—the odds are that novel words are more likely to appear in legitimate mail than in junk.

`--pdiag` *fname*
: Write a diagnostic file to the specified *fname* containing the actual lines the parser processed (after decoding of MIME parts and exclusion of data deemed unparseable). Use this option when you suspect problems in decoding or pre-parser filtering.

`--phraselimit` *n*
: Limit the length of phrases assembled according to the `--phrasemin` and `--phrasemax` options to *n* characters. This permits ignoring "phrases" consisting of gibberish from mail headers and un-decoded content. In most cases these items will be discarded by a `--prune` in any case, but skipping them as they are generated keeps the dictionary from bloating in the first place. The default value is 0 characters, which enforces no limit on phrase length.

`--phrasemin` *n*
: Calculate probabilities of phrases consisting of a minumum of *n* words. The default of 1 calculates probabilities for single words.

`--phrasemax` *n*
: Calculate probabilities of phrases consisting of a maximum of *n* words. The default of 1 calculates probabilities for single words. If you set this too large, the dictionary may grow to an absurd size.

`--prune`
: After loading the dictionary from `--mail` and `--junk` folders, this option discards words which appear sufficiently infrequently that their probability cannot be reliably estimated. One usually `--prune`s the dictionary before using `--write` to save it for subsequent runs.

`--plot` *fname*
: After loading the dictionary, create a plot in *fname*`.png` of the histogram of words, binned by their probability of appearance in junk mail. In order to generate the histogram the `GNUPLOT` and `NETPbm` utilities must be installed on the system; if they are absent, the `--plot` option will not be available.

`--ptrace`
: Include a token-by-token trace in the `--pdiag` output file. This helps when adjusting the parser's criteria for recognising tokens. Setting this option without also specifying a `--pdiag` file will have no effect other than perhaps to exercise your fingers typing it on the command line.

`--read`, `-r` *fname*
: Load a dictionary (previously created with the `--write` option) from file *fname.*

`--sigwords` *n*
: The probability that a message is junk will be computed based on the individual probabilities of the *n* words with extremal probabilities; that is, probabilities most indicative of junk or mail. The default is 15, but there's no obvious optimal setting for this parameter; it depends in part on the average length of messages you receive.

`--statistics`
: After loading the dictionary from `--mail` and `--junk` folders, print statistics of the distribution of junk probabilities of words in the dictionary. The statistics are written to standard output.

`--test`, `-t` *fname*
: Test mail in *fname* and write the estimated probability it is junk to standard output unless the `--transcript` option is also specified with standard output ("`-`") as the destination, in which case the inclusion of the probability and classification in the transcript is adjudged sufficient. If the `--verbose` option is specified, the individual probabilities of the "most interesting" words in the message will also be output. If *fname* is "`-`" the message is read from standard input.

`--threshjunk` *n*
: Set the threshold for classifying a message as junk to the floating point probability value *n*. The default threshold is 0.9; messages scored above `--threshjunk` are deemed junk.

`--threshmail` *n*
: Set the threshold for classifying a message as legitimate mail to the floating point probability value *n*. The default threshold is 0.9, with messages scored below `--threshmail` deemed legitimate. Note that you may leave a gap between the `--threshmail` and `--threshjunk` values (although it makes no sense to set `--threshmail` higher). Mail scored between the two threshold will then be judged of uncertain status.

`--transcript` *fname*
: Write an annotated transcript of the original message to the specified *fname*. If *fname* is "`-`", the transcript is written to standard output. At the end of the message header, an `X-Annoyance-Filter-Junk-Probability` header item giving the computed probability and an `X-Annoyance-Filter-Classification` item which gives the classification of the message according to the `--threshmail` and `--threshjunk` settings; the classification is given as "`Mail`", "`Junk`", or "`Indeterminate`".

`--verbose`, `-v`
: Print diagnostic information as the program performs various operations.

`--version`
: Print program version information.

`--write` *fname*
: Write a dictionary to the file *fname*. The dictionary is written in a binary format which may be loaded on subsequent runs with the `--read` option. Binary dictionary files are portable among machines with different architectures and byte ordering.

5. Integrating with Procmail.

Many UNIX users plagued by junk mail already use the Procmail program to filter incoming mail. Procmail makes it easy to define a "whitelist" of senders whose mail is always of interest and a "blacklist" of known perpetrators of junk mail. Although Procmail includes a flexible weighted scoring mechanism for evaluating mail based on content, this has limitations in coping with real world junk mail. First of all, choosing keywords and their scores is a completely manual process which requires continual attention as the content of junk mail evolves. Trial and error is the only mechanism to avoid "false positives" (legitimate mail erroneously considered junk) and "false negatives" (junk which makes it through the filter). Further, Procmail looks only at the raw message received by the mail agent, and contains no logic to decode attachments, parse HTML, or interpret encoded character sets. Present-day junk mail has these attributes in profusion, and often deliberately employs them in the interest of "stealth"—evading keyword based filters such as Procmail.

annoyance-filter has been designed to work either stand-alone or in conjunction with a filter like Procmail. Integrating annoyance-filter and Procmail provides the best of both worlds—hand-crafted Procmail filtering of the obvious cases (whitelists, blacklists, and routine mail filing) and annoyance-filter evaluation of the unclassified residua. Here's how you can go about integrating annoyance-filter and Procmail. In the examples below, we'll use "blohard" as the user name of the person installing annoyance-filter. ■

5.1. Installing annoyance-filter

First of all, you need to build annoyance-filter for your system, create a dictionary from collections of legitimate and junk mail, and install the lot in a location where the mail transfer agent (Sendmail on most UNIX systems) can access it. This can be any directory owned by the user, but I recommend you use the default of .annoyance-filter in your home ($HOME) directory; this is the destination used by the install target in the Makefile.

After you've built your custom dictionary, copy it to the .annoyance-filter directory as dict.bin.

5.2. Installing Procmail

Obviously, if you're going to be using Procmail, it needs to be installed on your system. Fortunately, many present-day Linux distributions come with Procmail already installed, so all the user need do is place the filtering rules (or "recipes") in a .procmailrc file in the home directory. If Procmail is not installed on your system, please visit Procmail for details on how to remedy that lacuna. If you do need to install Procmail, note that it can be installed either system-wide, filtering all users' mail (this is how the Linux distributions generally install it), or on a per-user basis, which does not require super-user permissions to install. Fortunately, the configuration file is identical regardless of how Procmail is installed.

5.3. Procmail Configuration

The next few paragraphs will look at typical components of a Procmail configuration file which, by default, is .procmailrc in the user's home directory. To make the script more generic and portable, we'll start by defining a few environment variables which specify where Procmail files mail and writes its log.

```
MAILDIR=$HOME/mailbox    # Be sure this directory exists
LOGFILE=$MAILDIR/logfile # Write a log of Procmail's actions
```

5.3.1. Filtering with annoyance-filter

annoyance-filter integrated with Procmail as a *filter*. As each message arrives, Procmail feeds it through annoyance-filter, which appends its estimation of the probability the message is junk to the header of the message. Subsequent Procmail recipes then test this field and route the message accordingly.

Assuming you've installed annoyance-filter in the $HOME/annoyance-filter directory, you activate the filtering by adding the following lines to your .procmailrc file. If you make this the first recipe, any subsequent recipe will be able to test for the annoyance-filter header fields.

```
:0 fw
| $HOME/.annoyance-filter/annoyance-filter \
      --read $HOME/.annoyance-filter/dict.bin --trans - --test -
```

The action line which pipes the message to `annoyance-filter` is continued onto a second line here in order to fit on the page. `Procmail` permits continuations of this form, but will equally accept the command all on one line with the backslash removed.

5.3.2. Routing by `annoyance-filter` classification

Once the message has been filtered by `annoyance-filter`, subsequent rules can test for its classification and route the message accordingly. The following rules dispatch messages it classifies as junk to a `junk` folder used by the blacklist, while messages judged to be legitimate mail and those with an intermediate probability are sent to the user's mailbox. (With the default settings, `annoyance-filter` will always classify a message as mail or junk, but if the `--threshjunk` and `--threshmail` settings are changed to as to create a gap between them, intermediate classification can occur.) Actually, the latter two recipes could be omitted since any message which fails to trigger any `Procmail` rule is sent to the user's mailbox by default. The variable `$ORGMAIL` is defined by `Procmail` as the user's mailbox; using it avoids using the specific path name which is dependent on the user name and mail system configuration.

```
:0 H:
* ^X-Annoyance-Filter-Classification: Junk
junk

:0 H:
* ^X-Annoyance-Filter-Classification: Mail
$ORGMAIL

:0 H:
* ^X-Annoyance-Filter-Classification: Indeterminate
$ORGMAIL
```

Even if you set the mail and junk probabilities so that messages can be classified as "`Indeterminate`", you're unlikely to see many so categorised—as long as the collections of mail and junk you used to train `annoyance-filter` are sufficiently large and representative, the vast majority of messages will usually be scored near the extremes of probability. If you're seeing a lot of `Indeterminate` messages, you should sort them manually, add them to the appropriate collection, and re-train `annoyance-filter`.

If you have other `Procmail` recipes for handling specific categories of mail, you would normally place the `annoyance-filter` related recipes *after* them, at the very end of the `procmailrc` file. That way `annoyance-filter`'s evalution is used as the final guardian at the gate before a message is delivered to your mailbox.

5.3.3. Other useful `.procmailrc` rules

The following subsections have nothing at all to do with `annoyance-filter`, really. You can set up a `.procmailrc` file based exclusively on `annoyance-filter` classifications as described above. Still, in many cases a few `Procmail` rules are worthwile in addition to `annoyance-filter` filtering. Here are some frequently used categories. You would normally place these rules *before* the `annoyance-filter` rules discussed in section 3.2.

5.3.3.1. Whitelist

Most people have a short list of folks with whom they correspond regularly. It's embarrassing if the content of a message from one of them is mistakenly identified as junk mail. To prevent this, define a "whitelist" as the first rule in your `Procmail` configuration after the filter command; messages which match its patterns avoid further scrutiny and are delivered directly to your mailbox. You should generally include your own address in the whitelist, as well as addresses of administrative accounts on machines you're responsible for, but be careful: junk mailers increasingly use sender addresses such as `root` to exploit whitelists. Here's user `blohard`'s whitelist definition. Multiple `Procmail` rules are normally combined with a logical AND ($\wedge$) operation. Since the whitelist requires an OR ($\vee$) operation, we manufacture one by a trivial application of `Procmail`'s weighted scoring facilities. `Procmail` patterns are regular expressions identical to those used by `egrep`, so metacharacters such as "`.`" must be quoted to be treated literally in patterns.

```
:0
* 0^0
*   1^1 ^From.*blohard@spectre\.org
*   1^1 ^From.*auric@spectre\.org
*   1^1 ^From.*bond@universal-impex\.co\.uk
*   1^1 ^From.*root@spectre\.org
$ORGMAIL
```

5.3.3.2. Blacklist

A "blacklist" works precisely like the whitelist, except that anything which matches one of its patterns is dispatched to the `junk` mail folder (or, if you're particularly confident there will be no false positives, to oblivion at `/dev/null`). Here we list some egregious spewers and unambiguous earmarks of junk mail. Note that in some cases it makes sense to match on header fields other than "`From`". By default, `Procmail`'s pattern matching is case-insensitive.

```
:0
* 0^0
*   1^1 ^From.*@link3buy\.com
*   1^1 ^From.*@lowspeedmediaoffers\.com
*   1^1 ^Subject:.*Let's be friends
*   1^1 ^X-Advertisement
*   1^1 ^X-Mailer.*RotMailer
*   1^1 ^To:.*Undisclosed.*Recipient
*   1^1 ^Subject:.*\[ADV\]
*   1^1 ^Subject:.*\(ADV\)
*   1^1 ^Reply-to:.*remove.*@
*   1^1 ^To.*friend
junk
```

At first glance, blacklists look like a good idea, but junk mail senders constantly change their domain names, and trigger words continually evolve protective colouration, making blacklist maintenance an never-ending process.

5.3.3.3. Automatic Filing

If you receive routine mail which you prefer to review as a batch from time to time, for example, messages from a mailing list to which you subscribe, you can have `Procmail` recognise them and file them in a folder for your eventual perusal. Obviously, you'll need to identify a pattern which matches all the messages in the category you wish to file but no others.

```
:0:
* ^From.*SUPER-VILLAINS +mailing +list
villains

:0 H:
* ^Subject.*Bacula: Backup OK
backups
```

Here, the user has provided a rule which files messages from a mailing list in a folder and notifications of successful backup completions (but not error notifications) from Bacula in a second folder.

6. To-do list.

- Translation of Chinese and Japanese characters currently decoded by the `GB2312` and `Big5` interpreters into their Unicode representations would permit uniform recognition of characters across the encodings.
- "Chinese junk" also sails into the harbour in the form of HTML in which the only indication of the character set is in a `charset=` declaration in the HTML itself, usually in a `http-equiv="Content-Type"` declaration. We ought to try to spot these and invoke the appropriate interpreter.
- Audit the MIME parsing code against RFCs 2045–2049 and subsequent updates (2231, 2387, 2557, 2646, and 3032, plus doubtless others). Examine various messages in the training collections which report MIME parsing and/or decoding errors to determine whether the messages are, indeed, malformed or are indicative of errors in this program.

6.1. Belling the cat

Most of the items on the above list require expertise I have not had the opportunity to acquire and/or research and experimentation I've lacked the time to perform. If you've the requisite knowledge for one or more of these jobs and are willing to put coding stick to magnetic domains, please get in touch. You can contact me by sending E-mail to `bugs@fourmilab.ch` with `annoyance-filter` in the `Subject` line.tmp/af.html

7. A Brief History of `annoyance-filter`.

In a real sense, this program has been twenty-five years in the making. The seed was planted in the 1970's while thinking about Jim Warren's concept of "datacasting". He envisioned using subcarriers of FM stations (or perhaps data encoded in the vertical retrace interval of television signals) to transmit digital information freely accessible to all. Not Xanadu or the Internet, mind you ... this remained a one-to-many broadcast medium, but one capable of providing information in a form which the then-emerging personal computers could receive, digest, and present in a customised fashion to their users.

"But who pays?" Well, that detail, which played a large part in the inflation and demise of the recent `.com` bubble, was central to the feasibility of datacasting as well. Jim Warren's view was that the primarily advertiser-supported business model adopted by most U.S. print and broadcast media would be equally applicable to bits flung into the ether from a radio antenna. As I recall, he cited the experience of suburban weekly newspapers, which discovered their profits *increased* when they moved from a paid subscription/per-copy readership to free distribution—circulation went up, advertising rates rose apace, and the bottom line changed from red to green.

Intriguing ... but still I had my doubts. When you read a newspaper or magazine, you can't avoid the advertising—you can flip past it, to be sure, but you still have to look at it, at least momentarily, so there's always the possibility a sufficiently clever image or tag line may motivate you to read the rest. I asked Jim why, once a document was in an entirely digital form, folks couldn't develop filters to remove the advertising before it ever reached their eyes. This would destroy the free distribution model and render an advertising-supported digital broadcasting service unworkable. Jim wasn't too concerned about this. In his estimation, discriminating advertising from editorial content would require artificial intelligence which did not exist and wasn't remotely on the horizon.

That's when von Mises' words on advertising came back to me. Advertising is *advertising*—perforce, it speaks with a *different vocabulary* than the sports page, letters to the editor, police blotter, national and international news, and commentary (aside, perhaps, from Maureen Dowd's columns in *The New York Times*). Given a sufficiently large collection of known editorial copy and advertising, might it not be possible to extract a *signature*, in the sense of radar signatures to discriminate warheads from decoys in ballistic missile defence, with which a sufficiently clever program could identify advertising and remove it, with a high level of confidence, before the reader ever saw it?

Fast forward—or, more precisely, *pause*.... By the late 1970's I'd concluded the best strategy to make the most of the ambient malaise was to amass a *huge pile* of money. Money may not buy happiness, but at the very least it would mitigate many of the irritations of that bleak, collectivist era. Being a nerd, I immediately turned to technology for a quick fix, and what should I espy but an exploding market in affordable home video cassette recorders—-VCRs—which were, in those days, becoming a fixture in more and more households. Many VCRs were purchased to play rented movies, but, being also able to automatically record programs off-the-air on a preset schedule, they could be used for "time-shifting"—recording broadcast programs for later viewing. But why, thought I, sit though all those tedious commercials you've recorded along with the programs you intend to watch? Certainly, people quickly learned to "zip"—use the fast forward to skip past commercials—but what if you could detect commercials and "zap" them—never record them in the first place? It occurred to me that inventing a device which accomplished this might be lucrative indeed.

The concept couldn't have been simpler—a little box which monitors the video and audio of the channel you're recording and, based on real-time analysis of the signal, pauses and resumes recording of the program on your VCR, yielding a tape free of advertising. It was easy to imagine such a gizmo succeeding like the contemporary "Demon Dialer" telephone speed dialer add-on, selling in the tens of millions in a matter of months.[2] Imagine the dismay of advertisers and my own contented avarice as I watched the money bin fill

[2] Well *of course* it occurred to me that widespread adoption of such a device would motivate advertisers to disguise the tags that discriminated commercials from programs. But hey—by the time that happened I'd have already cashed the customers' checks and blown the joint. There was bit of the Ferengi in me then. Truth be told, there still is.

deep enough for high diving. No more laps round the worry room for me!

I must confess to some inside information in this regard. While working for a regrettable employer in an odious swamp, I'd twigged to the fact that network television advertisers tagged their commercials with a signature in the vertical retrace interval to permit audit bureaux to measure how many network affiliates actually broadcast each commercial. This tag appeared to me the Achilles' heel of television advertising. As long as one could distinguish tagged commercials from an un-tagged program, it would be more or less straightforward to detect when a commercial was being transmitted and pause the VCR until the program resumed.

If only. . . . In reality, only nationally broadcast commercials bore the tag, and only some of them. Local commercials were never tagged. This created a difficult marketing dilemma for my grand scheme. While it might have been possible to block some of the most ubiquitous and irritating commercials on mass-market network series, the bottom feeders who *watch* those shows probably *enjoyed* the commercials and wouldn't be prospects for my gadget, while those like myself, infuriated by incessant commercials interrupting late night movies, would find the device ineffective since local commercials on independent stations were never tagged. Real-time analysis of video or even audio in the 1970's and early 80's was technologically out of the question for a product aimed at a mass consumer market. So, I put the idea of an annoyance filter for television aside and occupied myself with other endeavours.

We now arrive at the late 1980's. I'd spent the last decade or so filling up the money bin more or less flat out, and having reached a level I judged more than adequate, I began to turn my attention to matters I'd neglected during those laser-focused years.

Writing science fiction, for one thing. There was something about the advertising filter which had dug its way into my brain so deeply that nothing could dislodge it. The year is 1989; the Berlin Wall is about to tumble; and I'm scribbling a story about two programmers spending the downtime between Christmas and New Year's Day (the period when I'd accomplished about half of my own productive work over the previous half decade) prowling the nascent Internet for evidence of an extraterrestrial message already received, but not recognised as such. In

We'll Return, After this Message,

it is an *annoyance filter* which recognises an extraterrestrial message for what it is, *advertising*, and as von Mises observed, distinguishable by its own strident clamouring for attention.

A decade later, in the very years in which I set my science fiction story, I launched my own search for a message from our Creator hidden in the most obvious of locations—no results so far. Yet still I scour the Net.

Which brings us, more or less, to the present. The idea of an annoyance filter continued to intermittently occupy my thoughts, especially as the volume of junk arriving in my mailbox incessantly mounted despite ongoing efforts to filter it with increasingly voluminous and clever `Procmail` rules. Then, in August 2002, my friend and colleague Kern Sibbald brought to my attention Paul Graham's brilliant design for an adaptable, Bayesian filter to discriminate junk and legitimate mail by word frequencies measured in actual samples of mail pre-sorted into those categories. Now *that* sounded promising! Here was a design which was simple in concept, theoretically sound, and best of all, *it seemed to work.* Graham implemented his prototype filter in the "Arc" Lisp dialect used in his research. I decided to build a deployable tool in industrial-strength C++, founded on his design, and handling all the details required so the filter could, as much as possible, interpret mail the same way a human would—decoding, translating, and extracting wherever necessary to defeat the techniques junk mailers adopt to hide their content from nave filtering utilities.

This is not a simple task. Consider—you can probably sort out a message you're interested in reading from unsolicited junk in a fraction of a second, but that assumes it's presented to you after all of the mail transfer and content encodings have been peeled away to reveal the true colours of the content. Long gone are days when E-mail was predominantly ASCII text. Today, it's more than likely to be HTML (if not a Flash animation or some other horror), often transmitted in `Quoted-Printable` or `Base64` encodings largely in the interest of "stealth"—to hide the content from filters not equipped with the decoding facilities of a full-fledged mail client.

The `annoyance-filter` is based on Graham's crystalline vision of Bayesian scoring of messages by empirically determined word probabilities. It includes the tedious but essential machinery required to parse MIME multi-part mail attachments, decode non-plain-text parts, and interpret character sets in languages the user isn't accustomed to reading. This makes for great snowdrifts of software, but fortunately few details about which the typical user need fret.

Preliminary tests indicate `annoyance-filter` is inordinately effective in discriminating legitimate from junk mail. But this entire endeavour remains very much an active area of research and, consequently, `annoyance-filter` has been implemented as a toolkit intended to facilitate experiments with various filtering strategies and measuring the characteristics which best identify mail worth reading. You're more than welcome to build and install the program using the cookbook instructions but, if you're inclined to delve deeper, feel free to jump in—the programming's fine! Everyone is invited to contribute their own wisdom and creativity toward bringing to an end this intellectual pollution. Remember, when nobody ever sees junk mail, nobody will bother to send it. Let us commence rowing toward that happy landfall.

8. Dictionary Word.

A *dictionaryWord* represents a unique token found in an input stream. The *text* field is the **string** value of the token.

⟨ Class definitions 8 ⟩ ≡

```
class dictionaryWord {
public:
  static const unsigned int nCategories = 2;
  enum mailCategory {
    Mail = 0, Junk = 1, Unknown
  };
  string text;    /* The word itself */
  unsigned int occurrences[nCategories];    /* Number of occurrences in Mail and Junk */
  double junkProbability;    /* Probability this word appears in Junk */
  dictionaryWord(string s = "")
  {
    set(s);
  }
  void set(string s = "", unsigned int s_Mail = 0, unsigned int s_Junk = 0, double jProb = −1)
  {
    text = s;
    occurrences[Mail] = s_Mail;
    occurrences[Junk] = s_Junk;
    junkProbability = jProb;
  }
  string get(void) const
  {
    return text;
  }
  unsigned int n_mail(void) const
  {
    return occurrences[Mail];
  }
  unsigned int n_junk(void) const
  {
    return occurrences[Junk];
  }
  void add(mailCategory cat, unsigned int howMany = 1)
  {
    assert(cat ≡ Mail ∨ cat ≡ Junk);
    occurrences[cat] += howMany;
  }    /* Reset occurrences in category. Returns number of occurrences remaining in other categories.
         */
  unsigned int resetCat(mailCategory cat)
  {
    assert(cat ≡ Mail ∨ cat ≡ Junk);
    occurrences[cat] = 0;
    return occurrences[Mail] + occurrences[Junk];
  }
  void computeJunkProbability(unsigned int nMailMessages, unsigned int nJunkMessages, double
      mailBias = 2, unsigned int minOccurrences = 5);
  double getJunkProbability(void) const
  {
    return junkProbability;
  }
  unsigned int length(void) const
  {     /* Return length of word */
    return text.length();
  }
  void toLower(void)
  {     /* Convert to lower case */
    transform(text.begin(), text.end(), text.begin(), &dictionaryWord::to_iso_lower);
  }
  void describe(ostream &os = cout);
  void exportCSV(ostream &os = cout);
  bool importCSV(istream &is = cin);
  static string categoryName(mailCategory c)
  {
    return (c ≡ Mail) ? "mail" : ((c ≡ Junk) ? "junk" : "unknown");
  }
  void exportToBinaryFile(ostream &os);
  bool importFromBinaryFile(istream &is);
protected:
  ⟨ Transformation functions for algorithms 16 ⟩;
};
```

See also sections 17, 30, 36, 37, 38, 48, 58, 60, 62, 64, 69, 70, 72, 74, 77, 80, 81, 82, 84, 85, 87, 89, 103, 114, 118, 155, 158, 168, and 171.

This code is used in section 204.

9. In order to store **dictionaryWord** objects in ordered containers such as **map**, we must define the < operator. It ranks objects by lexical comparison of their *text* fields.

```
⟨ Class implementations 9 ⟩ ≡
  bool operator < (dictionaryWord a, dictionaryWord b)
  {
    return a.get() < b.get();
  }
```

See also sections 10, 11, 12, 13, 14, 15, 18, 19, 20, 21, 22, 23, 24, 25, 28, 29, 31, 32, 34, 35, 39, 46, 47, 49, 51, 54, 55, 59, 61, 63, 65, 71, 73, 75, 78, 79, 83, 86, 88, 90, 91, 92, 93, 94, 95, 96, 98, 104, 115, 119, 120, 125, 126, 149, 150, 151, 153, 154, 156, 157, 159, 166, 170, and 177.

This code is used in section 204.

10. The *computeJunkProbability* procedure determines the probability a given **dictionaryWord** appears in junk mail. Words with a high probability (near 1) are almost certain to be from junk, while low probability words (near 0) are highly likely to appear in legitimate mail. The probability is computed based on the following parameters:

m	*occurrences*[*Mail*]	Occurrences of word in legitimate mail
j	*occurrences*[*Junk*]	Occurrences of word in in junk mail
n_m	*nMailMessages*	Number of legitimate mail messages in database
n_j	*nJunkMessages*	Number of junk mail messages in database
b	*mailBias*	Bias in favour of words in legitimate messages
s	*minOccurrences*	Significance: discard words with $(m \times b + j) < s$

$$p = \begin{cases} -1, & \text{if } (m \times b + j) < s; \\ \min(0.99, \max(0.01, \frac{\min(j/n_j,1)}{\min((m\times b)/n_m,1)+\min(j/n_j,1))}) & \text{otherwise.} \end{cases}$$

A word which appears so few times its probability is deemed insufficiently determined is assigned a notional probability of −1 and ignored in subsequent tests. To avoid dividing by zero when incrementally assembling dictionaries, if no messages in a category have been loaded, we arbitrarily set the count to 1.

```
⟨ Class implementations 9 ⟩ +≡
  void dictionaryWord :: computeJunkProbability(unsigned int nMailMessages, unsigned int
          nJunkMessages, double mailBias, unsigned int minOccurrences)
  {
    double nMail = occurrences[Mail] ∗ mailBias, nJunk = occurrences[Junk];
    nMailMessages = max(nMailMessages, 1U);
    nJunkMessages = max(nJunkMessages, 1U);
    if ((nMail + nJunk) ≥ minOccurrences) {
      assert(nMailMessages > 0);
      assert(nJunkMessages > 0);
      junkProbability = min(0.99, max(0.01, min(nJunk/nJunkMessages,
          1.0)/(min(nMail/nMailMessages, 1.0) + min(nJunk/nJunkMessages, 1.0))));
    }
    else {
      junkProbability = −1;
    }
  }
```

11. The *describe* method writes a human-readable description of the various fields in the object to the designated output stream, which defaults to *cout*.

```
⟨ Class implementations 9 ⟩ +≡
  void dictionaryWord :: describe(ostream &os)
  {
    os ≪ text ≪ "␣␣Mail:␣" ≪ n_mail() ≪ ",␣Junk:␣" ≪ n_junk() ≪ ",␣Probability:␣" ≪
        setprecision(5) ≪ junkProbability ≪ endl;
  }
```

12. The *exportCSV* method creates a comma-separated value (CSV) file containing all fields from the dictionary word. This permitting verification and debugging of the dictionary compilation process.

```
⟨ Class implementations 9 ⟩ +≡
  void dictionaryWord :: exportCSV (ostream &os)
  {
    os ≪ setprecision(5) ≪ junkProbability ≪ "," ≪ occurrences[Mail] ≪ "," ≪ occurrences[Junk] ≪
        ",\"" ≪ text ≪ "\"" ≪ endl;
  }
```

13. The *importCSV* method reads the next line from a comma-separated value (CSV) dictionary dump and stores the values parsed from it into the **dictionaryWord**. If this is the special sentinel pseudo-word used to store the message counts, *junkProbability* will be set to −1. If the record is not a well-formed CSV dictionary word, *junkProbability* will be set to −2 and *text* to the actual line from the CSV file; this may be used to discard title records. Records which begin with "`;`" or "`#`" are ignored as comments. When the end of file is encountered, *false* is returned and *junkProbability* is set to −3.

Note that this is *not* a general purpose CSV parser, but rather one specific to the format which *exportCSV* writes. In particular, general string quoting is ignored since none of the difficult cases arise in the CSV we generate.

```
⟨ Class implementations 9 ⟩ +≡
  bool dictionaryWord :: importCSV(istream &is = cin)
  {
    while (true) {
      string s;
      if (getline(is, s)) {
        string :: size_typep, p1, p2;
        for (p = 0; p < s.length(); p++) {
          if (¬isISOspace(s[p])) {
            break;
          }
        }
        if ((p ≥ s.length()) ∨ (s[p] ≡ '#') ∨ (s[p] ≡ ';')) {
          continue;     /* Blank line or comment delimiter—ignore */
        }
        if ((s[p] ≡ '-') ∨ isdigit(s[p])) {
          p = s.find(',');
          if (p ≠ string :: npos) {
            p1 = s.find(',', p + 1);
            if (p1 ≠ string :: npos) {
              p2 = s.find(',', p1 + 1);
              if (p2 ≠ string :: npos) {
                junkProbability = atof(s.substr(0, p).c_str());
                occurrences[Mail] = atoi(s.substr(p + 1, p1 − p).c_str());
                occurrences[Junk] = atoi(s.substr(p1 + 1, p2 − p).c_str());
                p = s.find('"', p2 + 1);
                if (p ≠ string :: npos) {
                  p1 = s.find_last_of('"');
                  if ((p1 ≠ string :: npos) ∧ (p1 > p)) {
                    text = s.substr(p + 1, (p1 − p) − 1);
                    return true;     /* A valid record, hurrah! */
                  }
                }
              }
            }
          }
        }
        junkProbability = −2;     /* Ill-formed record */
        text = s;
        return true;
      }
      junkProbability = −3;     /* End of file */
      return false;
    }
  }
```

14. This method writes a binary representation of the word to an output stream. This is used to create the binary word database used to avoid rebuilding the letter and character category counts every time. Each entry begins with the number of characters in the word followed by its text. After this, the count and probability fields are output in portable big-endian format. We do assume IEEE floating point compatibility across platforms, but auto-detect floating point byte order.

```
⟨ Class implementations 9 ⟩ +≡
  void dictionaryWord :: exportToBinaryFile(ostream &os){ unsigned char c;
        const unsigned char *fp;
        const double k1 = −1.0;
#define outCount(x)assert(x ≤ 255);
        c = (x); os.put(c)
#define outNumber(x)os.put((x ≫ 24) & #FF);
        os.put((x ≫ 16) & #FF);
        os.put((x ≫ 8) & #FF); os.put(x & #FF)
        outCount(text.length( ));
        os.write(text.data( ), text.length( ));
        outNumber(n_mail( ));
        outNumber(n_junk( ));
        fp = reinterpret_cast⟨const unsigned char *⟩(&k1);
        if (fp[0] ≡ 0) {
          fp = reinterpret_cast⟨unsigned char *⟩(&junkProbability);
          for (unsigned int i = 0; i < (sizeof junkProbability); i++) {
            outCount(fp[((sizeof junkProbability) − 1) − i]);
          }
        }
        else {      /* Big-endian platform */
          os.write(&junkProbability, sizeof junkProbability);
        }
#undef outCount
#undef outNumber
        }
```

15. Importing a word from a binary file is the inverse of the export above. Once again we figure out the byte order of **double** on the fly by testing a constant and decode the byte stream accordingly.

```
⟨ Class implementations 9 ⟩ +≡
  bool dictionaryWord :: importFromBinaryFile(istream &is)
  {
    unsigned char c;
    char sval[256];
    unsigned char ibyte[4];
    unsigned char fb[8];
    unsigned char *fp;
    const double k1 = −1.0;
    const unsigned char *kp;
#define iNumber   ((ibyte[0] ≪ 24) | (ibyte[1] ≪ 16) | (ibyte[2] ≪ 8) | ibyte[3])
    if (is.read(&c, 1)) {
      if (is.read(sval, c)) {
        text = string(sval, c);
        is.read(ibyte, 4);
        occurrences[Mail] = iNumber;
        is.read(ibyte, 4);
        occurrences[Junk] = iNumber;
        kp = reinterpret_cast⟨const unsigned char *⟩(&k1);
        if (kp[0] ≡ 0) {
          is.read(fb, 8);
          fp = reinterpret_cast⟨unsigned char *⟩(&junkProbability);
          for (unsigned int i = 0; i < (sizeof junkProbability); i++) {
            fp[((sizeof junkProbability) − 1) − i] = fb[i];
          }
        }
        else {
          is.read(&junkProbability, sizeof junkProbability);
        }
        return true;
      }
    }
    return false;
#undef iNumber
  }
```

16. The following are simple-minded transformation functions passed as arguments to STL algorithms for various manipulations of the text.

```
⟨ Transformation functions for algorithms 16 ⟩ ≡
  static char to_iso_lower(char c)
  {
    return toISOlower(c);
  }
  static char to_iso_upper(char c)
  {
    return toISOupper(c);
  }
```

This code is used in section 8.

17. Dictionary.
A *dictionary* is a collection of **dictionaryWord** objects, organised for rapid look-up. For convenience and efficiency, we derive *dictionary* from the STL **map** container, thereby making all of its core functionality accessible to the user. It would be more efficient and cleaner to use a **set**, but objects in a **set** cannot be modified; values in a **map** can.

```
⟨ Class definitions 8 ⟩ +≡
  class dictionary : public map⟨string, dictionaryWord⟩ {
public:
  void add(dictionaryWord w, dictionaryWord :: mailCategory category); void
  include (dictionaryWord &w) ;
  void exportCSV(ostream &os = cout);
  void importCSV(istream &is = cin);
  void computeJunkProbability(unsigned int nMailMessages, unsigned int nJunkMessages, double
      mailBias = 2, unsigned int minOccurrences = 5);
  void purge(void);
  void resetCat(dictionaryWord :: mailCategory category);
  void printStatistics(ostream &os = cout) const;
#ifdef HAVE_PLOT_UTILITIES
  void plotProbabilityHistogram(string fileName, unsigned int nBins = 20) const;
#endif
  void exportToBinaryFile(ostream &os);
  void importFromBinaryFile(istream &is); } ;
```

18. The *add* method looks up a **dictionaryWord** in the **dictionary**. If the word is already present, its number of occurrences in the given *category* is incremented. Otherwise, the word is added to the **dictionary** with the occurrence count for the *category* initialised to 1.

```
⟨ Class implementations 9 ⟩ +≡
  void dictionary :: add(dictionaryWord w, dictionaryWord :: mailCategory category)
  {
    dictionary :: iterator p;
    if ((p = find(w.get( ))) ≠ end( )) {
      p→second.add(category);
    }
    else {
      insert(make_pair(w.get( ), w)).first→second.add(category);
    }
  }
```

19. The **include** method is used when merging dictionaries, for example when performing an *importFromBinaryFile*. It looks up the argument word in the dictionary. If present, its occurrence counts are added to those of the existing word. Otherwise, a new word is added with the occurence counts of the argument.

⟨ Class implementations 9 ⟩ +≡

```
void dictionary::include (dictionaryWord &w)
{
  dictionary::iterator p;
  if ((p = find(w.get())) ≠ end()) {
    p→second.occurrences[dictionaryWord::Mail] += w.occurrences[dictionaryWord::Mail];
    p→second.occurrences[dictionaryWord::Junk] += w.occurrences[dictionaryWord::Junk];
  }
  else {
    insert(make_pair(w.get(), w));
  }
}
```

20. The *exportCSV* method exports the dictionary in comma-separated value (CSV) format for debugging. To simplify analysis, the dictionary is re-sorted by *junkProbability*. The *byProbability* comparison function is introduced to permit this sorting of the dictionary. A pseudo-word is added at the start of the CSV file to give the number of mail and junk messages scanned in preparing it.

⟨ Class implementations 9 ⟩ +≡

```
bool byProbability(const dictionaryWord *w1, const dictionaryWord *w2)
{
  double dp = w1→getJunkProbability() − w2→getJunkProbability();
  if (dp ≡ 0) {
    return w1→get() < w2→get();
  }
  return dp < 0;
}
void dictionary::exportCSV (ostream &os)
{
  if (verbose) {
    cerr ≪ "Exporting␣dictionary␣to␣CSV␣file." ≪ endl;
  }
  vector⟨dictionaryWord *⟩ dv;
  for (iterator p = begin(); p ≠ end(); p++) {
    dv.push_back(&(p→second));
  }
  sort(dv.begin(), dv.end(), byProbability);
  os ≪ ";␣Probability,Mail,Junk,Word" ≪ endl;
  dictionaryWord pdw;
  pdw.set(pseudoCountsWord, messageCount[dictionaryWord::Mail],
      messageCount[dictionaryWord::Junk], −1);
  pdw.exportCSV(os);
  for (vector⟨dictionaryWord *⟩::iterator q = dv.begin(); q ≠ dv.end(); q++) {
    (*q)→exportCSV(os);
  }
}
```

21. We import a dictionary from a CSV file by importing successive records into a **dictionaryWord**, which is then appended to the **dictionary**. When the pseudo-word containing the number of mail and junk messages used to assemble the dictionary is encountered, those quantities are added to the running totals. Note that the CSV input file may be in any order—it need not be sorted in the order *exportCSV* creates, nor need the message count pseudo-word be the first record of the file.

```
⟨ Class implementations 9 ⟩ +≡
  void dictionary::importCSV(istream &is)
  {
    if (verbose) {
      cerr ≪ "Importing␣dictionary␣from␣CSV␣file." ≪ endl;
    }
    dictionaryWord dw;
    while (dw.importCSV(is)) {
      if (dw.getJunkProbability() ≡ −1 ∧ (dw.get() ≡ pseudoCountsWord)) {
        messageCount[dictionaryWord::Mail] += dw.n_mail();
        messageCount[dictionaryWord::Junk] += dw.n_junk();
      }
      else if (dw.getJunkProbability() ≥ −1) {
        include (dw) ;
      }
      else {
        if (verbose) {
          cerr ≪ "Ill-formed␣record␣in␣CSV␣import:␣\"" ≪ dw.get() ≪ "\"" ≪ endl;
        }
      }
    }
  }
```

22. The *purge* method discards words in the dictionary which occur sufficiently infrequently that no probability has been assigned them. May I say a few words about how we accomplish this? Yes, it looks absurd to move the elements we wish to preserve to a separate **queue**, then transfer them back once we're done emptying the **map**. "Why not just walk through the items and *erase* any which don't make the cut?", you ask. Because you *can't*, I reply. Performing an *erase* on a **map** invalidates all iterators to it, so once you've removed an item, you're forced to restart the scan from the *begin*( ) iterator; with a large dictionary to purge, that takes *forever*.

Now STL purists will observe that I ought be using the *remove_if* algorithm rather than iterating over the container myself. Well, if you can figure out how to make it work, you're a better man than I. I defined a predicate to perform a less test on the probability of the **dictionaryWord** in the second part of the **pair**, and this contraption makes it past the compiler intact. But when I attempt to pass that predicate to *remove_if* I get half a page of gibberish from the bowels of STL complaining about not being able to use the default assignment operator on **string pair**⟨**const string**, **dictionaryWord**⟩ :: *first* or some such. If you can figure out how to make this work, be my guest—I'll be glad to replace my code with yours with complete attribution. I've left my *remove_if* code (which doesn't make it through the compiler) below, disabled on the tag `PURGE_USES_REMOVE_IF`. Good luck—me, I'm finished.

"A man is not finished when he is defeated. He is finished when he quits."
—Richard M. Nixon

```
⟨ Class implementations 9 ⟩ +≡
#ifdef PURGE_USES_REMOVE_IF
  class dictionaryWordProb_less : public unary_function < pair⟨string, dictionaryWord⟩ , int >
  {
    int p;
  public:
    explicit dictionaryWordProb_less(const int pt)
    : p(pt) { }
    bool operator( )(const pair⟨string, dictionaryWord⟩ &dw) const
    {
      return dw.second.getJunkProbability( ) < p;
    }
  }
  ;
#endif
  void dictionary :: purge(void)
  {
    if (verbose) {
      cerr ≪ "Pruning␣rare␣words␣from␣database:␣" ≪ flush;
    }
#ifdef PURGE_USES_REMOVE_IF
    remove_if(begin( ), end( ), dictionaryWordProb_less(0));
#else
    queue⟨dictionaryWord⟩ pq;
    while (¬empty( )) {
      if (begin( )→second.getJunkProbability( ) ≥ 0) {
        pq.push(begin( )→second);
      }
      erase(begin( ));
    }
    while (¬pq.empty( )) {
      insert(make_pair(pq.front( ).get( ), pq.front( )));
```

```
        pq.pop();
      }
#endif
      if (verbose) {
        cerr ≪ "␣words␣remaining." ≪ endl;
      }
    }
```

23. The *resetCat* method resets the count for all words for the given **mailCategory**.

```
⟨ Class implementations 9 ⟩ +≡
  void dictionary::resetCat(dictionaryWord::mailCategory category)
  {
    if (verbose) {
      cerr ≪ "Resetting␣counts␣for␣category␣" ≪ dictionaryWord::categoryName(category) ≪
          endl;
    }
    for (iterator mp = begin(); mp ≠ end(); mp++) {
      mp→second.resetCat(category);
    }
  }
```

24. Compute and print statistical measures of the probability distribution of words in the dictionary. Words with negative probability are ignored, so there is no need to *purge* before computing statistics.

```
⟨ Class implementations 9 ⟩ +≡
  void dictionary::printStatistics(ostream &os) const{
      if (verbose) {
        cerr ≪ "Computing␣dictionary␣statistics." ≪ endl;
      }
      os ≪ "Dictionary␣statistics:" ≪ endl; dataTable < double > dt;
      for (const_iterator mp = begin(); mp ≠ end(); mp++) {
        if (mp→second.getJunkProbability() ≥ 0) {
          dt.push_back(mp→second.getJunkProbability());
        }
      }
      os ≪ "Mean␣=␣" ≪ dt.mean() ≪ endl;
      os ≪ "Geometric␣mean␣=␣" ≪ dt.geometricMean() ≪ endl;
      os ≪ "Harmonic␣mean␣=␣" ≪ dt.harmonicMean() ≪ endl;
      os ≪ "RMS␣=␣" ≪ dt.RMS() ≪ endl;
      os ≪ "Median␣=␣" ≪ dt.median() ≪ endl;
      os ≪ "Mode␣=␣" ≪ dt.mode() ≪ endl;
      os ≪ "Percentile(0.5)␣=␣" ≪ dt.percentile(0.5) ≪ endl;
      os ≪ "Quartile(1)␣=␣" ≪ dt.quartile(1) ≪ endl;
      os ≪ "Quartile(3)␣=␣" ≪ dt.quartile(3) ≪ endl;
      os ≪ "Variance␣=␣" ≪ dt.variance() ≪ endl;
      os ≪ "Standard␣deviation␣=␣" ≪ dt.stdev() ≪ endl;
      os ≪ "CentralMoment(3)␣=␣" ≪ dt.centralMoment(3) ≪ endl;
      os ≪ "Skewness␣=␣" ≪ dt.skewness() ≪ endl;
      os ≪ "Kurtosis␣=␣" ≪ dt.kurtosis() ≪ endl; }
```

25. Plot a histogram of the distribution of words in the dictionary by probability. Words with negative probability are ignored, so there is no need to *purge* before plotting.

```
⟨ Class implementations 9 ⟩ +≡
#ifdef HAVE_PLOT_UTILITIES
#define PLOT_DEBUG
  void dictionary::plotProbabilityHistogram(string fileName, unsigned int nBins) const
  {
    if (verbose) {
      cerr ≪ "Plotting␣probability␣histogram␣to␣" ≪ fileName ≪ ".png" ≪ endl;
    }
    ofstream gp((fileName + ".gp").c_str()), dat((fileName + ".dat").c_str());
    ⟨ Build histogram of word probabilities 26 ⟩;
    ⟨ Write GNUPLOT data table for probability histogram 27 ⟩;
      /* Create GNUPLOT instructions to plot data */
    gp ≪ "set␣term␣pbm␣small␣color" ≪ endl;
    gp ≪ "set␣ylabel␣\"Number␣of␣Words\"" ≪ endl;
    gp ≪ "set␣xlabel␣\"Probability\"" ≪ endl;
    gp ≪ "plot␣\"" ≪ fileName ≪ ".dat\"␣using␣1:2␣title␣\"\"␣with␣boxes" ≪ endl;
    string command("gnuplot␣");
    command += fileName + ".gp␣|␣pnmtopng␣>" + fileName + ".png";
#ifdef PLOT_DEBUG
    cout ≪ command ≪ endl;
#else
    command += "␣2>/dev/null";
#endif
    gp.close();
    dat.close();
    system(command.c_str());
#ifndef PLOT_DEBUG      /* Delete the temporary files used to create the plot */
    remove((fileName + ".gp").c_str());
    remove((fileName + ".dat").c_str());
#endif
  }
#endif     /* HAVE_PLOT_UTILITIES */
```

26. Walk through the dictionary and bin the probabilities of words into *nBins* equally sized bins and compute a histogram of the numbers in each bin.

```
⟨ Build histogram of word probabilities 26 ⟩ ≡
  vector⟨unsigned int⟩ hist(nBins);
  for (const_iterator mp = begin(); mp ≠ end(); mp++) {
    if (mp→second.getJunkProbability() ≥ 0) {
      unsigned int bin = static_cast⟨unsigned int⟩(mp→second.getJunkProbability() * nBins);
      hist[bin]++;
    }
  }
```

This code is used in section 25.

27. Write the `GNUPLOT` data file for the probability histogram. The first field in each line is the binned probability and the second is the number of words which fell into that bin.

⟨ Write GNUPLOT data table for probability histogram 27 ⟩ ≡
```
for (unsigned int j = 0; j < nBins; j++) {
   dat ≪ (static_cast⟨double⟩(j)/nBins) ≪ "␣" ≪ hist[j] ≪ endl;
}
```
This code is used in section 25.

28. When the dictionary has been modified, recompute the junk probability of all the words it contains. This simply applies the *computeJunkProbability* method to all the **dictionaryWord**s in the container.

⟨ Class implementations 9 ⟩ +≡
```
void dictionary :: computeJunkProbability(unsigned int nMailMessages, unsigned int
         nJunkMessages, double mailBias, unsigned int minOccurrences)
{
   for (dictionary :: iterator p = begin(); p ≠ end(); p++) {
      p→second.computeJunkProbability(nMailMessages, nJunkMessages, mailBias, minOccurrences);
   }
}
```

29. Exporting or importing a dictionary to or from a binary file is more or less a matter of iterating through the dictionary and delegating the matter to each individual word. One detail we must deal with, however, is adding a pseudo-word at the head of the dictionary to record the number of mail and junk *messages* which contributed the words to the dictionary. These counts are needed to subsequently recompute the probability for each word.

When loading a dictionary with *importFromBinaryFile* this pseudo-word is recognised and the values it contains are added to the *messageCount* for each category. Note that importing a file is logically an *addition* to an existing dictionary—you may import any number of binary dictionary files, just as you can add mail folders with the `--mail` and `--junk` options.

```
#define pseudoCountsWord "␣COUNTS␣"
⟨ Class implementations 9 ⟩ +≡
  void dictionary :: exportToBinaryFile(ostream &os)
  {
    if (verbose) {
      cerr ≪ "Exporting␣dictionary␣to␣binary␣file." ≪ endl;
    }
    dictionaryWord pdw;
    pdw.set(pseudoCountsWord, messageCount[dictionaryWord :: Mail],
        messageCount[dictionaryWord :: Junk], −1);
    pdw.exportToBinaryFile(os);
    for (dictionary :: iterator p = begin( ); p ≠ end( ); p++) {
      p→second.exportToBinaryFile(os);
    }
  }
  void dictionary :: importFromBinaryFile(istream &is)
  {
    if (verbose) {
      cerr ≪ "Importing␣dictionary␣from␣binary␣file." ≪ endl;
    }
    dictionaryWord dw;
    if (dw.importFromBinaryFile(is)) {
      assert(dw.get( ) ≡ pseudoCountsWord);
      messageCount[dictionaryWord :: Mail] += dw.n_mail( );
      messageCount[dictionaryWord :: Junk] += dw.n_junk( );
      while (dw.importFromBinaryFile(is)) {
        include (dw) ;
      }
    }
  }
```

30. MIME decoders.
MIME decoders process parts of multi-part messages in various MIME encodings such as `base64` and `Quoted-Printable`. They read encoded lines from an **istream** and return decoded binary values with the *getchar* method. The decoder terminates when the current MIME *partBoundary* is encountered.
MIMEdecoder is the parent class of all specific decoders.

```
⟨ Class definitions 8 ⟩ +≡
  class mailFolder;
  class MIMEdecoder {
  public:
    istream *is;    /* Stream from which encoded lines are read */
    string partBoundary;    /* Part boundary sentinel */
    bool atEnd;    /* At end of part or stream ? */
    bool eofHit;    /* Was decoder terminated by end of file ? */
    unsigned int nDecodeErrors;    /* Number of decoding errors */
  protected:
    string inputLine;    /* Current encoded input line */
    string::size_type ip;    /* Input line pointer */
    unsigned encodedLineCount;    /* Number of encoded lines read */
    bool lookAhead;    /* Have we looked ahead ? */
    int lookChar;    /* Look-ahead character */
    string endBoundary;    /* Terminating part boundary */
    list⟨string⟩ *tlist;    /* Transcript list */
    mailFolder *mf;    /* Parent mail folder */
  public:
    MIMEdecoder(istream *i = Λ, mailFolder *m = Λ, string pb = "", list⟨string⟩ *tl = Λ)
    {
      set(i, m, pb, tl);
      resetDecodeErrors();
      tlist = Λ;
    }
    virtual ~MIMEdecoder()
    { }
    ;
    void set(istream *i = Λ, mailFolder *m = Λ, string pb = "", list⟨string⟩ *tl = Λ)
    {
      is = i;
      mf = m;
      partBoundary = pb;
      inputLine = "";
      ip = 0;
      encodedLineCount = 0;
      lookAhead = false;
      atEnd = false;
      eofHit = false;
      tlist = tl;
    }
    virtual string name(void) const = 0;
    virtual void resetDecodeErrors(void)
    {
      nDecodeErrors = 0;
    }
    virtual unsigned int getDecodeErrors(void) const
    {
      return nDecodeErrors;
    }
    virtual string getTerminatorSentinel(void) const
    {
      return endBoundary;
    }
    virtual bool isEndOfFile(void) const
    {
      return eofHit;
    }
    virtual unsigned int getEncodedLineCount(void) const
    {
      return encodedLineCount;
    }
    virtual int getDecodedChar(void) = 0;      /* Return next decoded character, < 0 if EOF */
    virtual bool getDecodedLine(string &s);
      /* Return next decoded line, return false for EOF */
    virtual void saveDecodedStream(ostream &os);      /* Write decoded text to an ostream */
    virtual void saveDecodedStream(const string fname);
      /* Write decoded text to file fname */
  protected:
    virtual bool getNextEncodedLine(void);
  };
```

31. The *getNextEncodedLine* method is called by specific decoders to obtain the next line (all encodings are line-oriented, being intended for inclusion in mail messages). The line is stored into *inputLine* and tested against the MIME part boundary sentinel. A logical end of file is reported when the part boundary is encountered. The method is declared **virtual** so derived decoders may override it if different behaviour is required.

One subtlety is that decoders may also be activated to decode the main body of a message. In this case, the *partBoundary* is set to the null string and body content is decoded until the start of the next message is encountered.

⟨ Class implementations 9 ⟩ +≡

```
bool MIMEdecoder :: getNextEncodedLine(void)
{
  if (¬atEnd) {
    if (getline(*is, inputLine) ≠ Λ) {
      if (inputLine.substr(0, (sizeof messageSentinel) − 1) ≡ messageSentinel) {
        endBoundary = inputLine;
        if (partBoundary ≠ "") {
          assert(mf ≠ Λ);
          mf→reportParserDiagnostic("Unterminated␣MIME␣sentinel␣at␣end␣of␣message.");
        }
        atEnd = true;
      }
      if ((partBoundary ≠ "") ∧ (inputLine.substr(0, 2) ≡ "--") ∧ (inputLine.substr(2,
            partBoundary.length( )) ≡ partBoundary)) {
        if (Annotate('d')) {
          ostringstream os;
          os ≪ "Part␣boundary␣encountered:␣" ≪ inputLine;
          mf→reportParserDiagnostic(os);
        }
        endBoundary = inputLine;
        atEnd = true;
      }
      else {
        if (tlist ≠ Λ) {
          tlist→push_back(inputLine);
        }
        ip = 0;
        encodedLineCount++;
      }
    }
    else {
      atEnd = true;
      eofHit = true;
    }
  }
  if (atEnd) {
    inputLine = "";
    ip = 0;
  }
  return ¬atEnd;
}
```

32. We provide a default implementation of *getDecodedLine* for derived classes. This forms lines from calls on *getDecodedChar*, accepting (and discarding) end of line sequences.

```
⟨ Class implementations 9 ⟩ +≡
  bool MIMEdecoder :: getDecodedLine(string &s)
  {
    int ch;
    s = "";
    while (true) {
      if (lookAhead) {
        ch = lookChar;
        lookAhead = false;
      }
      else {
        ch = getDecodedChar( );
      }
      if (ch < 0) {
        break;
      }
      ⟨ Check for and process end of line sequence 33 ⟩;
      s += ch;
    }
    return s.length( ) > 0;
  }
```

33. In order to support all plausible end of line sequences, we need to look ahead one character at end of line; if the caller intends to intermix calls on *getDecodedLine* and *getDecodedChar* (a pretty dopey thing to do, it must be said), the *getDecodedChar* implementation in the derived class must be aware that look ahead may have happened and properly interact with the *lookAhead* flag.

```
⟨ Check for and process end of line sequence 33 ⟩ ≡
  if (ch ≡ '\r' ∨ ch ≡ '\n') {
    int cht = getDecodedChar( );
    if (¬(((ch ≡ '\r') ∧ (cht ≡ '\n')) ∨ ((ch ≡ '\n') ∧ (cht ≡ '\r')))) {
      lookAhead = true;
      lookChar = cht;
    }
    return true;
  }
```

This code is used in section 32.

34. We may want to export a decoded part to a file or, perhaps, save it as a string stream for further examination. This method writes decoded bytes to its **ostream** argument.

```
⟨ Class implementations 9 ⟩ +≡
  void MIMEdecoder :: saveDecodedStream(ostream &os)
  {
    int ch;
    while ((ch = getDecodedChar( )) ≥ 0) {
      os.put(ch);
    }
  }
```

35. We also provide a flavour of *saveDecodedStream* which exports the decoded stream to a named file.

```
⟨ Class implementations 9 ⟩ +≡
  void MIMEdecoder :: saveDecodedStream(const string fname)
  {
    ofstream of(fname.c_str());
    if (¬of) {
      if (verbose) {
        cerr ≪ "Cannot␣create␣MIMEdecoder␣dump␣file:␣" ≪ fname ≪ endl;
      }
    }
    else {
      saveDecodedStream(of);
      of.close();
    }
  }
```

36. Identity MIME decoder.

The *identityMIMEdecoder* is a trivial MIME decoder which simply passes through text in the part without transformation. It is provided as a test case and template for genuinely useful decoders. It may also come in handy should the need arise for the interposition of an obligatory decoder even for MIME parts which can be read directly as text.

```
⟨ Class definitions 8 ⟩ +≡
  class identityMIMEdecoder : public MIMEdecoder {
  public:
    string name(void) const
    {
      return "Identity";
    }
    int getDecodedChar(void)
    {
      while (¬atEnd) {
        if (ip < inputLine.length()) {
          return inputLine[ip++] & #FF;
        }
        if (getNextEncodedLine()) {
          continue;
        }
      }
      return −1;
    }
    bool getDecodedLine(string &s)
    {
      if (ip < inputLine.length()) {
        s = inputLine.substr(ip);
        ip = inputLine.length();
        return true;
      }
      if (getNextEncodedLine()) {
        s = inputLine;
        ip = inputLine.length();
        return true;
      }
      return false;
    }
  };
```

37. Sink MIME decoder.

The *sinkMIMEdecoder* simply discards lines from the MIME part the first time *getDecodedChar* or *getDecodedLine* is called. It is used for skipping parts in which we aren't interested.

```
⟨ Class definitions 8 ⟩ +≡
  class sinkMIMEdecoder : public MIMEdecoder {
  public:
    string name(void) const
    {
      return "Sink";
    }
    int getDecodedChar(void)
    {
      if (¬atEnd) {
        while (getNextEncodedLine()) ;
        assert(atEnd);
      }
      return −1;
    }
  };
```

38. Base64 MIME decoder.

The base64MIMEdecoder decodes an input stream encoded as MIME `base64` per RFC 1341. This is based on my stand-alone `base64` decoder.

```
⟨ Class definitions 8 ⟩ +≡
  class base64MIMEdecoder : public MIMEdecoder {
  private:
    unsigned char dtable[256];    /* Decoding table */
    void initialiseDecodingTable(void);    /* Initialise decoding table */
    deque⟨unsigned char⟩ decodedBytes;    /* Decoded bytes queue */
  public:
    base64MIMEdecoder( )
    {
      initialiseDecodingTable( );
    }
    string name(void) const
    {
      return "Base64";
    }
    int getDecodedChar(void);
    static string decodeEscapedText(const string s, mailFolder *m = Λ);
  };
```

39. The *getDecodedChar* returns decoded characters from the *decodedBytes* queue, refilling it with triples of bytes decoded from the input stream as required. When the end of the stream is encountered, −1 is returned.

```
⟨ Class implementations 9 ⟩ +≡
  int base64MIMEdecoder :: getDecodedChar(void)
  {
    ⟨ Check for look ahead character 45 ⟩;
    if (decodedBytes.size( ) ≡ 0) {
      ⟨ Refill decoded bytes queue from input stream 40 ⟩;
    }
    if (decodedBytes.size( ) > 0) {
      unsigned char v = decodedBytes[0];
      decodedBytes.pop_front( );
      return v;
    }
    return −1;
  }
```

40. This is the heart of the `base64` decoder. It reads the next four significant (non-white space) characters from the input stream, extracts the 6 bits encoded by each, and assembles the bits into three 8 bit bytes which are added to the *decodedBytes* queue. Although the current decoder always immediately empties the queue, in principal any sequence of the encoded content up to its entire length may be decoded by repeated invocations of this code.

```
⟨ Refill decoded bytes queue from input stream 40 ⟩ ≡
  unsigned char a[4], b[4], o[3];
  int j, k;
  ⟨ Decode next four characters from input stream 41 ⟩;
  ⟨ Assemble the decoded bits into bytes and place on decoded queue 44 ⟩;
```

This code is used in section 39.

41. Read the next four non-blank bytes from the input stream, checking for end of file, and place their decoded 6 bit values into the array *b*. We save the original encoded characters in array *a* to permit testing them for the special "`=`" sentinel which denotes short sequences at the end of file.

```
⟨ Decode next four characters from input stream 41 ⟩ ≡
  for (int i = 0; i < 4; i++) {
    int c;
    ⟨ Get next significant character from input stream 42 ⟩;
    ⟨ Check for end of file in base64 stream 43 ⟩;
    if (dtable[c] & #80) {
      nDecodeErrors++;
      ostringstream os;
      os ≪ "Illegal␣character␣'" ≪ c ≪ "'␣in␣Base64␣input␣stream.";
      mf→reportParserDiagnostic(os.str());     /* Ignoring errors: discard invalid character. */
      i−−;
      continue;
    }
    a[i] = (unsigned char) c;
    b[i] = dtable[c];
  }
```

This code is used in section 40.

42. Read the encoded input stream and return the next non-white space character. This code does not verify whether characters it returns are valid within a `base64` stream—that's up to the caller to determine once the character is returned.

```
⟨ Get next significant character from input stream 42 ⟩ ≡
  while (true) {
    c = −1;
    while (ip < inputLine.length()) {
      if (inputLine[ip] > '␣') {
        c = inputLine[ip++];
        break;
      }
      ip++;
    }
    if (c ≥ 0) {
      break;
    }
    if (¬getNextEncodedLine()) {
      break;
    }
  }
```

This code is used in section 41.

43. An end of file indication (due to encountering the MIME part separator sentinel) is valid only after an even number of four character encoded sequences. Validate this and report any errors accordingly. If an unexpected end of file is encountered, any incomplete encoded sequence is discarded.

```
⟨ Check for end of file in base64 stream 43 ⟩ ≡
  if (c ≡ EOF) {
    if (i > 0) {
      nDecodeErrors++;
      mf→reportParserDiagnostic("Unexpected␣end␣of␣file␣in␣Base64␣decoding.");
    }
    return −1;
  }
```

This code is used in section 41.

44. Once we've decoded four characters from the input stream, we have four six-bit fields in the *b* array. Now we extract, shift, and ∨ these fields together to form three 8 bit bytes. One subtlety arises at the end of file. The last one or two characters of an encoded four character field may be replaced by equal signs to indicate that the final field encodes only one or two source bytes. If this is the case, the number of bytes placed onto the *decodedBytes* queue is reduced to the correct value.

```
⟨ Assemble the decoded bits into bytes and place on decoded queue 44 ⟩ ≡
  o[0] = (b[0] ≪ 2) | (b[1] ≫ 4);
  o[1] = (b[1] ≪ 4) | (b[2] ≫ 2);
  o[2] = (b[2] ≪ 6) | b[3];
  j = a[2] ≡ '=' ? 1 : (a[3] ≡ '=' ? 2 : 3);
  for (k = 0; k < j; k++) {
    decodedBytes.push_back(o[k]);
  }
```

This code is used in section 40.

45. Since we rely on the parent class default implementation of *getNextEncodedLine*, if we wish to permit intermixed calls on *getNextEncodedLine* and *getNextEncodedChar* we must cope with the fact that the last *getNextEncodedLine* call may have peeked ahead one character. If so, clear the look ahead flag and return the look ahead character.

```
⟨ Check for look ahead character 45 ⟩ ≡
  if (lookAhead) {
    lookAhead = false;
    return lookChar;
  }
```

This code is used in sections 39 and 49.

46. The *initialiseDecodingTable* method fills the binary encoding table with the characters the 6 bit values are mapped into. The curious and disparate sequences used to fill this table permit this code to work both on ASCII and EBCDIC systems.

In EBCDIC systems character codes for letters are not consecutive; the initialisation must be split to accommodate the EBCDIC consecutive letters:

`A–I J–R S–Z a–i j–r s–z`

This code works on ASCII as well as EBCDIC systems.

```
⟨ Class implementations 9 ⟩ +≡
  void base64MIMEdecoder :: initialiseDecodingTable(void)
  {
    int i;
    for (i = 0; i < 255; i++) {
      dtable[i] = #80;
    }
    for (i = 'A'; i ≤ 'I'; i++) {
      dtable[i] = 0 + (i − 'A');
    }
    for (i = 'J'; i ≤ 'R'; i++) {
      dtable[i] = 9 + (i − 'J');
    }
    for (i = 'S'; i ≤ 'Z'; i++) {
      dtable[i] = 18 + (i − 'S');
    }
    for (i = 'a'; i ≤ 'i'; i++) {
      dtable[i] = 26 + (i − 'a');
    }
    for (i = 'j'; i ≤ 'r'; i++) {
      dtable[i] = 35 + (i − 'j');
    }
    for (i = 's'; i ≤ 'z'; i++) {
      dtable[i] = 44 + (i − 's');
    }
    for (i = '0'; i ≤ '9'; i++) {
      dtable[i] = 52 + (i − '0');
    }
    dtable['+'] = 62;
    dtable['/'] = 63;
    dtable['='] = 0;
  }
```

47. The **static** method *decodeEscapedText* decodes text in its **string** argument, returning a string with escape sequences replaced by the encoded characters. Note that, notwithstanding this being a **static** method which can be invoked without reference to a **base64MIMEdecoder** object, we in fact actually instantiate such an object within the method, supplying its input from an **istringstream** constructed from the argument **string**.

```
⟨ Class implementations 9 ⟩ +≡
  string base64MIMEdecoder :: decodeEscapedText(const string s, mailFolder *m = Λ)
  {
    string r = "";
    base64MIMEdecoder dc;
    istringstream iss(s);
    int dchar;

    dc.set(&iss, m, "");
    while ((dchar = dc.getDecodedChar()) ≥ 0) {
      r += static_cast⟨char⟩(dchar);
    }
    return r;
  }
```

48. Quoted-Printable MIME decoder.
The quotedPrintableMIMEdecoder decodes an input stream encoded as MIME “Quoted-Printable” per RFC 1521. This is based on my stand-alone Quoted-Printable decoder.

⟨ Class definitions 8 ⟩ +≡

```
class quotedPrintableMIMEdecoder : public MIMEdecoder {
public:
  quotedPrintableMIMEdecoder( )
  {
    atEndOfLine = false;
  }
  string name(void) const
  {
    return "Quoted-Printable";
  }
  int getDecodedChar(void);
  static string decodeEscapedText(const string s, mailFolder *m = Λ);
protected:
  bool atEndOfLine;
  int getNextChar(void);
  static int hex_to_nybble(const int ch);
};
```

49. Get the next decoded character from the stream, expanding “=” escape sequences.

⟨ Class implementations 9 ⟩ +≡

```
int quotedPrintableMIMEdecoder :: getDecodedChar(void)
{
  int ch;
  ⟨ Check for look ahead character 45 ⟩;
  while (true) {
    ch = getNextChar( );
    if (ch ≡ '=') {
      ⟨ Decode equal sign escape 50 ⟩;
    }
    else {
      return ch;
    }
  }
}
```

50. When we encounter an equal sign in the input stream there are two possibilities: it may introduce two characters of ASCII representing an 8-bit octet in hexadecimal or, if followed by an end of line sequence, it's a "soft end-of-line" introduced to avoid emitting a line longer than the maximum number of characters prescribed by the RFC.

```
⟨ Decode equal sign escape 50 ⟩ ≡
  int ch1 = getNextChar( );
  ⟨ Ignore white space after soft line break 53 ⟩;
  if (ch1 ≡ '\n') {
    continue;
  }
  else {
    int n1 = hex_to_nybble(ch1);
    int ch2 = getNextChar( );
    int n2 = hex_to_nybble(ch2);
    if (n1 ≡ −1 ∨ n2 ≡ −1) {
      ostringstream os;
      os ≪ "Invalid␣escape␣sequence␣'=" ≪ static_cast⟨char⟩(ch1) ≪
          static_cast⟨char⟩(ch2) ≪ "'␣in␣Quoted-Printable␣MIME␣part.";
      mf→reportParserDiagnostic(os.str( ));
      nDecodeErrors++;
    }
    ch = (n1 ≪ 4) | n2;
  }
  return ch;
```

This code is used in section 49.

51. Return the next character from the encoded input stream. Since end of line sequences have been stripped, we append our own new-line character to the end of each line. This indicates that in the absence of a soft line break (trailing equal sign), we should emit a line break to the output stream.

```
⟨ Class implementations 9 ⟩ +≡
  int quotedPrintableMIMEdecoder :: getNextChar(void)
  {
    while (true) {
      if (atEndOfLine) {
        atEndOfLine = false;
        return '\n';
      }
      if (ip < inputLine.length()) {
        if (ip ≡ (inputLine.length() − 1)) {
          atEndOfLine = true;
        }
        return inputLine[ip++];
      }
      if (¬getNextEncodedLine()) {
        break;
      }
      if (inputLine.length() ≡ 0) {
        atEndOfLine = true;
      }
    }
    return −1;
  }
```

52. There are lots of ways of defining “ASCII white space,” but RFC 1521 explicitly states that only ASCII space and horizontal tab characters are deemed white space for the purposes of Quoted-Printable encoding.

```
⟨ Character is white space 52 ⟩ ≡
  ((ch1 ≡ '␣') ∨ (ch1 ≡ '\t'))
```

This code is used in section 53.

53. Some systems pad text lines with white space (ASCII blank or horizontal tab characters). This may result in a line encoded with a "soft line break" at the end appearing, when decoded, with white space between the supposedly-trailing equal sign and the end of line sequence. If white space follows an equal sign escape, we ignore it up to the beginning of an end of line sequence. Non-white space appearing before we sense the end of line is an error; these erroneous characters are ignored.

```
⟨ Ignore white space after soft line break 53 ⟩ ≡
  while (⟨ Character is white space 52 ⟩) {
    ch1 = getNextChar( );
    if (ch1 ≡ '\n') {
      continue;
    }
    if (¬⟨ Character is white space 52 ⟩) {
      nDecodeErrors ++;
      ostringstream os;
      os ≪ "Invalid␣character␣'" ≪ static_cast⟨char⟩(ch1) ≪
          "'␣after␣soft␣line␣break␣in␣Quoted-Printable␣MIME␣part.";
      mf→reportParserDiagnostic(os.str( ));
      ch1 = '␣';      /* Fake a space and soldier on */
    }
  }
```

This code is used in section 50.

54. The *hex_to_nybble* method converts a hexadecimal digit in the sequence "0123456789ABCDEF" or the equivalent with lower case letters to its binary value. If an invalid hexadecimal digit is supplied, −1 is returned.

```
⟨ Class implementations 9 ⟩ +≡
  int quotedPrintableMIMEdecoder :: hex_to_nybble(const int ch)
  {
    if ((ch ≥ '0') ∧ (ch ≤ ('0' + 9))) {
      return ch − '0';
    }
    else if ((ch ≥ 'A') ∧ (ch ≤ ('A' + 5))) {
      return 10 + (ch − 'A');
    }
    else if ((ch ≥ 'a') ∧ (ch ≤ ('a' + 5))) {
      return 10 + (ch − 'a');
    }
    return −1;
  }
```

55. The **static** method *decodeEscapedText* decodes text in its **string** argument, returning a string with escape sequences replaced by the encoded characters.

⟨ Class implementations 9 ⟩ +≡
```
string quotedPrintableMIMEdecoder :: decodeEscapedText(const string s, mailFolder
          *m = Λ)
{
  string r = "";

  string :: size_type p;
  for (p = 0; p < s.length( ); p++) {
    bool decoded = false;

    if (s[p] ≡ '=') {
      if (p > (s.length( ) − 3)) {
        if (verbose) {
          cerr ≪ "decodeEscapedText:␣escape␣too␣near␣end␣of␣string:␣" ≪ s ≪ endl;
        }
      }
      else {
        int n1 = hex_to_nybble(s[p + 1]), n2 = hex_to_nybble(s[p + 2]);

        if ((n1 < 0) ∨ (n2 < 0)) {
          if (verbose) {
            cerr ≪ "decodeEscapedText:␣invalid␣escape␣sequence␣\"" ≪ s.substr(p,
                3) ≪ "\"" ≪ endl;
          }
        }
        else {
          r += static_cast⟨char⟩((n1 ≪ 4) | n2);
          decoded = true;
          p += 2;
        }
      }
    }
    if (¬decoded) {
      r += s[p];
    }
  }
  return r;
}
```

56. Multiple byte character set decoders and interpreters.

To support languages with character sets too large to be encoded in a single byte, a bewildering variety of *multiple byte character sets* are employed. In a rational world, there would be a single, universal, and uniform encoding of every glyph used in human written encoding, and a unique way of representing this in byte-oriented messages.

Rather amazingly, there *is* such a representation: ISO/IEC 10646 and its UTF-8 encoding. Not surprisingly, hardly anybody uses it—it's an international standard, after all. So, we must cope with a plethora of character sets and byte encodings, than that's the lot in life of the *MBCSdecoder* and *MBCSinterpreter*. These abstract classes are the parent of specific decoders for various encodings and interpreters for the motley crowd of character sets.

First, let's define our terms. A *decoder* is charged with chewing through a byte stream and identifying the logical characters within it, in all their various lengths. Decoders must cope with encoding such as EUC, shift-JIS, and UTF-8. An *interpreter*'s responsibility is expressing the character codes delivered by the decoder in a form comprehensible to those not endowed with the original language character set or knowledge of how to read it. This usually means encoding ideographic languages where each character more or less corresponds to a word as space-separated tokens uniquely identifying the character code (by its hexadecimal code, for example), and characters in word-oriented languages as unique strings which meet the downstream rules for tokens. For example, one might express a sequence of Chinese characters in the "`Big5`" character set as:

```
big5-A2FE big5-E094 big5-F3CA
```

or a two words in a Cyrillic font as:

```
cyr-A0cyr-98cyr-81cyr-FE cyr-84cyr-D3cyr-EAcyr-A7
```

(These examples were just made up off the cuff—if they represent something heroically obscene in some representation of a language, it's just my lucky day.)

Note that because of what we're doing here, we don't have to remotely comprehend the character set or read the language to be highly effective in accomplishing our mission. Like cryptographers who broke book codes without knowing the language of the plaintext, we're concerned only with the frequency with which various tokens, however defined, occur in legitimate and junk mail. As long as our representations are unique and more or less correspond to tokens in the underlying language, we don't need to understand what it *means*.

57. Decoders.

58. Decoder parent class.

This is the abstract parent class of all specific decoders. Albeit abstract in the details, we provide a variety of services to derived classes.

```
⟨ Class definitions 8 ⟩ +≡
  class MBCSdecoder {
  protected:
    const string *src;
    string::size_type p;
  public:
    virtual string name(void) = 0;    /* Name of decoder */
    virtual void setSource(const string &s)
    {    /* Set input source line */
      src = &s;
      p = 0;
    }
    virtual void reset(void)
    {    /* Reset stateful decoder to ground state */
    }
    virtual int getNextDecodedChar(void) = 0;    /* Get next decoded character */
    virtual int getNextEncodedByte(void)
    {
      if (p ≥ src→length( )) {
        return −1;
      }
      return ((*src)[p++]) & #FF;
    }
  protected:
    virtual int getNextNBytes(const unsigned int n);
    virtual int getNext2Bytes(void)
    {
      return getNextNBytes(2);
    }
    virtual int getNext3Bytes(void)
    {
      return getNextNBytes(3);
    }
    virtual int getNext4Bytes(void)
    {
      return getNextNBytes(4);
    }
    virtual void discardLine(void)
    {
      p = src→length( );
    }
  };
```

59. Return a character assembled by concatenating the next n bytes in most significant byte to least significant byte order. If the end of input is encountered, -1 is returned. A multiple byte character equal to -1 triggers an assertion failure in debug builds.

```
⟨ Class implementations 9 ⟩ +≡
  int MBCSdecoder::getNextNBytes(const unsigned int n)
  {
    assert((n ≥ 1) ∧ (n ≤ 4));
    int v = 0;
    for (unsigned int i = 0; i < n; i++) {
      int b = getNextEncodedByte( );
      if (b < 0) {
        return b;
      }
      v = (v ≪ 8) | b;
    }
    assert(v ≠ −1);
    return v;
  }
```

60. EUC decoder.
This decoder extracts logical characters from byte streams encoded in EUC encoding. In EUC, if a byte in the input stream is in the range #A1–#FE and the subsequent byte in the range #80–#FF, then the variant fields encoded in the two bytes define the character code. A byte not within the range of the first byte of a two byte character is interpreted as a single byte character with ASCII/ISO-8859 semantics.

```
⟨ Class definitions 8 ⟩ +≡
  class EUC_MBCSdecoder : public MBCSdecoder {
  public:
    virtual string name(void)
    {
      return "EUC";
    }
    virtual int getNextDecodedChar(void);      /∗ Get next decoded byte ∗/
  };
```

61. Bytes are parsed from the input stream as follows. Any bytes with values within the range #A1–#FE denote the first byte of a two byte character, whose second byte must be within the range #80–#FF. Any violation of the constraints on the second byte indicates an invalid sequence. Characters outside the range of initial characters are considered single byte codes. We return −1 when the end of the encoded line is encountered.

```
⟨Class implementations 9⟩ +≡
  int EUC_MBCSdecoder::getNextDecodedChar(void)
  {
    int c1 = getNextEncodedByte();
    if ((c1 ≥ #A1) ∧ (c1 ≤ #FE)) {
      int c2 = getNextEncodedByte();
      if ((c2 ≥ #80) ∧ (c2 ≤ #FF)) {
        return (c1 ≪ 8) | c2;
      }
      if (c2 ≡ −1) {
        if (verbose) {
          cerr ≪ name() ≪ "_MBCSdecoder:␣Premature␣end␣of␣line␣in␣\
              two␣byte␣character." ≪ endl;
          return −1;
        }
      }     /* Odds are that once we've encountered an invalid second byte, the balance of the
             encoded line will be screwed up as well. To avoid such blithering, discard the line after
             such an error. */
      discardLine();
      if (verbose) {
        cerr ≪ name() ≪ "_MBCSdecoder:␣Invalid␣second␣byte:␣0x" ≪ hex ≪ c2 ≪ dec ≪
            "␣in␣two␣byte␣character." ≪ endl;
      }
      return c1;
    }
    return c1;
  }
```

62. Big5 decoder.
This decoder extracts logical characters from byte streams encoded in Big5 encoding. In Big5, bytes in the range #00–#7F are single ASCII characters. Bytes with the #80 bit set are the first byte of a two byte character, the second byte of which may have any value.

```
⟨Class definitions 8⟩ +≡
  class Big5_MBCSdecoder : public MBCSdecoder {
  public:
    virtual string name(void)
    {
      return "Big5";
    }
    virtual int getNextDecodedChar(void);      /* Get next decoded byte */
  };
```

63. Decode the next logical character. We return −1 when the end of the encoded line is encountered.

```
⟨ Class implementations 9 ⟩ +≡
  int Big5_MBCSdecoder::getNextDecodedChar(void)
  {
    int c1 = getNextEncodedByte();
    if ((c1 ≥ 0) ∧ ((c1 & #80) ≠ 0)) {
      int c2 = getNextEncodedByte();
      if (c2 ≡ −1) {
        if (verbose) {
          cerr ≪ name() ≪ "_MBCSdecoder:␣Premature␣end␣of␣line␣in␣\
              two␣byte␣character." ≪ endl;
        }
        return −1;
      }
      return (c1 ≪ 8) | c2;
    }
    return c1;
  }
```

64. Shift-JIS decoder.

Shift-JIS is used to encode Japanese characters on MS-DOS, Windows, and the Macintosh (which adds four additional one-byte characters which we support here). The encoding uses code points #21–#7E for ASCII/JIS-Roman single byte characters, code points #A1–#DF for single byte hald width katakana, plus two-byte characters introduced by first bytes in the ranges #81–#9F, #E0–#EF, and, for user-defined characters, #F0–#FC. The second byte of a valid two-byte character will always be in one of the ranges #40–#7E and #80–#FC.

```
⟨ Class definitions 8 ⟩ +≡
  class Shift_JIS_MBCSdecoder : public MBCSdecoder {
  protected:
    string pending;
  public:
    Shift_JIS_MBCSdecoder()
    : pending("") { }
    virtual ∼Shift_JIS_MBCSdecoder()
    { }
    virtual string name(void)
    {
      return "Shift_JIS";
    }
    virtual int getNextDecodedChar(void);      /* Get next decoded byte */
  };
```

65. Decode the next logical character. We return −1 when the end of the encoded line is encountered. An invalid second byte of a two byte character terminates processing of the line, as it's likely to be gibberish from then on.

```
⟨ Class implementations 9 ⟩ +≡
  int Shift_JIS_MBCSdecoder :: getNextDecodedChar(void)
  {
    ⟨ Check for pending characters and return if so 67 ⟩;
    int c1 = getNextEncodedByte( );
    if (c1 ≥ 0) {
      ⟨ Check for Shift-JIS two byte character and assemble as required 66 ⟩;
      ⟨ Check for Macintosh-specific single byte characters and translate 68 ⟩;
    }
    return c1;
  }
```

66. We test for the first byte we've read being in the range which denotes a two byte character. If so, read the second byte of the character, validating that it is within the ranges permitted for second bytes, and assemble the 16 bit character from the two bytes.

```
⟨ Check for Shift-JIS two byte character and assemble as required 66 ⟩ ≡
  if (((c1 ≥ #81) ∧ (c1 ≤ #9F)) ∨ ((c1 ≥ #E0) ∧ (c1 ≤ #EF)) ∨ ((c1 ≥ #F0) ∧ (c1 ≤ #FC))) {
    int c2 = getNextEncodedByte( );
    if (c2 ≡ −1) {
      if (verbose) {
        cerr ≪ name( ) ≪ "_MBCSdecoder:␣Premature␣end␣of␣line␣in␣two␣byte␣character." ≪
            endl;
      }
      return −1;
    }
    if (¬(((c2 ≥ #40) ∧ (c2 ≤ #7E)) ∨ ((c2 ≥ #80) ∧ (c2 ≤ #FC)))) {
      if (verbose) {
        cerr ≪ name( ) ≪ "_MBCSdecoder:␣Invalid␣second␣byte␣in␣two␣byte␣character." ≪
            endl;
      }
      return −1;
    }
    return (c1 ≪ 8) | c2;
  }
```

This code is used in section 65.

67. To permit expansion of Macintosh-specific characters to multiple character replacements, we have the ability to store the balance of a multiple character sequence in the *pending* string. If there are any characters there, return them before obtaining another character from the input stream.

```
⟨ Check for pending characters and return if so 67 ⟩ ≡
  if (¬pending.empty( )) {
    int pc = pending[0];
    pending = pending.substr(1);
    return pc;
  }
```

This code is used in section 65.

68. The four additional characters added by the Macintosh are #80 (backslash), #FD (copyright symbol), #FE (trademark symbol), and #FF (ellipsis). We check for them and translate them into plausible ISO 8859 replacements, expanding as necessary into multiple character sequences via the *pending* string mechanism.

```
⟨ Check for Macintosh-specific single byte characters and translate 68 ⟩ ≡
  switch (c1) {
  case #80:
     c1 = '\\';     /* Macintosh backslash */
     break;
  case #FD:
     c1 = #A9;     /* ISO 8859 © symbol */
     break;
  case #FE: c1 = 'T';     /* Trademark (TM) symbol */
     pending = "M";
     break;
  case #FF:     /* Ellipsis ("...") */
     c1 = '.';
     pending = "..";
     break;
  }
```

This code is used in section 65.

69. Unicode decoders.

The Unicode character set (itself a subset of the 32 bit ISO 10646 character set), uses a variety of encoding schemes. The *Unicode_MBCSdecoder* is the parent class for all specific Unicode decoders and provides common services for them.

```
⟨ Class definitions 8 ⟩ +≡
  class Unicode_MBCSdecoder : public MBCSdecoder {
  public:
     virtual string name(void)
     {
        return "Unicode";
     }
     virtual int getNextDecodedChar(void) = 0;     /* Get next decoded byte */
  };
```

70. UCS-2 Unicode decoder.
UCS-2 encoding of Unicode is simply a sequence of 16 bit quantities, which may be stored in either little-endian or big-endian order; usually identified by a Unicode Byte Order Mark at the start of the file. Here we do not attempt to auto-sense byte order; it must be set by the setBigEndian method before the decoder is used.

```
⟨ Class definitions 8 ⟩ +≡
  class UCS_2_Unicode_MBCSdecoder : public Unicode_MBCSdecoder {
  protected:
    bool bigEndian;
  public:
    UCS_2_Unicode_MBCSdecoder(bool isBigEndian = true)
    {
      setBigEndian(isBigEndian);
    }
    void setBigEndian(bool isBigEndian = true)
    {
      bigEndian = isBigEndian;
    }
    virtual string name(void)
    {
      return "UCS_2_Unicode";
    }
    virtual int getNextDecodedChar(void);      /* Get next decoded byte */
  };
```

71. Decode the next logical character. We return −1 when the end of the encoded line is encountered.

```
⟨ Class implementations 9 ⟩ +≡
  int UCS_2_Unicode_MBCSdecoder::getNextDecodedChar(void)
  {
    int c1 = getNextEncodedByte();
    int c2 = getNextEncodedByte();
    if (c2 ≡ −1) {
      if (verbose) {
        cerr ≪ name() ≪ "_MBCSdecoder:␣Premature␣end␣of␣line␣in␣two␣byte␣character." ≪
            endl;
      }
      return −1;
    }
    if (bigEndian) {
      c1 = (c1 ≪ 8) | c2;
    }
    else {
      c1 |= (c2 ≪ 8);
    }
    return c1;
  }
```

72. UTF-8 Unicode decoder.

The UTF-8 encoding of Unicode is an ASCII-transparent encoding into a stream of 8 bit bytes. The length of encoded character is variable and forward-parseable.

⟨ Class definitions 8 ⟩ +≡

```
class UTF_8_Unicode_MBCSdecoder : public Unicode_MBCSdecoder {
public:
  virtual string name(void)
  {
    return "UTF_8_Unicode";
  }
  virtual int getNextDecodedChar(void);     /* Get next decoded byte */
};
```

73. Decode the next logical character. We return −1 when the end of the encoded line is encountered.

⟨ Class implementations 9 ⟩ +≡

```
int UTF_8_Unicode_MBCSdecoder::getNextDecodedChar(void)
{
  int c1 = getNextEncodedByte();
  if (c1 < 0) {
    return c1;    /* End of input stream */
  }
  string::size_type nbytes = 0;
  unsigned int result;
  if (c1 ≤ #7F) {    /* Fast track special case for ASCII 7 bit codes */
    result = c1;
    nbytes = 1;
  }
  else {
    unsigned char chn = c1;
        /* N.b. You can dramatically speed up the determination of how many bytes follow the
        first byte code by looking it up in a 256 byte table of lengths (with duplicate values as
        needed due to value bits in the low order positions. Once the length is determined, you can
        use a table look-up to obtain the mask for the first byte rather than developing the mask
        with a shift. The code which assembles the rest of the value could also be unrolled into
        individual cases to avoid loop overhead. Of course none of this is worth the bother unless
        you're going to be doing this a lot. */
    while ((chn & #80) ≠ 0) {
      nbytes++;
      chn ≪= 1;
    }
    if (nbytes > 6) {
      if (verbose) {
        cerr ≪ name() ≪ "_MBCSdecoder:␣Invalid␣first␣byte␣in␣UTF-8␣encoded␣string" ≪
            endl;
      }
      return −1;
    }
    result = c1 & (#FF ≫ (nbytes + 1));    /* Extract bits from first byte */
    for (string::size_type i = 1; i < nbytes; i++) {
      c1 = getNextEncodedByte();
      if (c1 < 0) {
        if (verbose) {
          cerr ≪ name() ≪ "_MBCSdecoder:␣Premature␣end␣of␣line␣in␣UTF-8␣character." ≪
              endl;
        }
        return −1;
      }
      if ((c1 & #C0) ≠ #80) {
        cerr ≪ name() ≪ "_MBCSdecoder:␣Bad␣byte␣1--n␣signature␣in␣UTF-8␣encoded␣sequ\
            ence" ≪ endl;
      }
      result = (result ≪ 6) | (c1 & #3F);
    }
  }
```

```
    return result;
  }
```

74. UTF-16 Unicode decoder.

The UTF-16 encoding of Unicode encodes logical characters as sequence of 16 bit codes. Most Unicode characters are encoded in a single 16 bit quantity, but character codes greater than 65535 are encoded in a pair of 16 bit values in the *surrogate* range. Naturally, this encoding can be either big- or little-endian in byte sequence; we handle either, as set by the *setBigEndian* method or the constructor.

⟨ Class definitions 8 ⟩ +≡

```
  class UTF_16_Unicode_MBCSdecoder : public Unicode_MBCSdecoder {
  protected:
    bool bigEndian;
    int getNextUTF_16Word(void)
    {
      int c1 = getNextEncodedByte();
      if (c1 < 0) {
        return c1;
      }
      int c2 = getNextEncodedByte();
      if (c2 < 0) {
        if (verbose) {
          cerr ≪ name() ≪ "_MBCSdecoder:␣Premature␣end␣of␣line␣in␣UTF-16␣character." ≪
              endl;
        }
        return −1;
      }
      if (bigEndian) {
        c1 = (c1 ≪ 8) | c2;
      }
      else {
        c1 |= (c2 ≪ 8);
      }
      return c1;
    }
  public:
    UTF_16_Unicode_MBCSdecoder(bool isBigEndian = true)
    {
      setBigEndian(isBigEndian);
    }
    void setBigEndian(bool isBigEndian = true)
    {
      bigEndian = isBigEndian;
    }
    virtual string name(void)
    {
      return "UTF_16_Unicode";
    }
    virtual int getNextDecodedChar(void);      /* Get next decoded byte */
  };
```

75. Decode the next logical character. We return −1 when the end of the encoded line is encountered.

⟨ Class implementations 9 ⟩ +≡

```
int UTF_16_Unicode_MBCSdecoder :: getNextDecodedChar(void)
{
  string :: size_type nwydes = 0;
  int w1, w2, result;
  w1 = getNextUTF_16Word( );
  if (w1 < 0) {
    return w1;
  }
  if ((w1 ≤ #D800) ∨ (w1 > #DFFF)) {
    result = w1;
    nwydes = 1;
  }
  else if ((w1 ≥ #D800) ∧ (w1 ≤ #DBFF)) {
    w2 = getNextUTF_16Word( );
    if (w2 < 0) {
      if (verbose) {
        cerr ≪ name( ) ≪ "_MBCSdecoder:␣Premature␣end␣of␣line␣in␣UTF-16␣two␣word␣char\
            acter." ≪ endl;
      }
      return −1;
    }
    nwydes = 2;
    if ((w2 < #DC00) ∨ (w2 > #DFFF)) {
      if (verbose) {
        cerr ≪ name( ) ≪ "_MBCSdecoder:␣Invalid␣second␣word␣surrogate␣in␣UTF-16␣encod\
            ed␣string." ≪ endl;
      }
      return −1;
    }
    result = (((w1 & #3FF) ≪ 10) | (w2 & #3FF)) + #10000;
  }
  else {
    if (verbose) {
      cerr ≪ name( ) ≪ "_MBCSdecoder:␣Invalid␣first␣word␣surrogate␣in␣UTF-16␣encode\
          d␣string." ≪ endl;
    }
    return −1;
  }
  return result;
}
```

76. Interpreters.

77. Interpreter parent class.
This is the abstract parent class of all concrete interpreters. We provide the services common to most decoders, while permitting them to be overridden by derived classes.

```
⟨ Class definitions 8 ⟩ +≡
  class MBCSinterpreter {
  protected:
    const string *src;
    MBCSdecoder *dp;
    string prefix, suffix;
  public:
    virtual ~MBCSinterpreter( )
    { }
    virtual string name(void) = 0;     /* Name of decoder */
    virtual void setDecoder(MBCSdecoder &d)
    {
      dp = &d;
    }
    virtual void setSource(const string &s)
    {     /* Set input source line */
      assert(dp ≠ Λ);
      dp→setSource(s);
    }
    virtual void setPrefixSuffix(string pre = "", string suf = "")
    {
      prefix = pre;
      suffix = suf;
    }
    virtual string getNextDecodedChar(void);
    virtual string decodeLine(const string &s);
  };
```

78. We provide this default implementation of *getNextDecodedChar* for derived classes. They're free to override it, but this may do the job for most. A logical character is obtained from the decoder. If its character code is less than or equal to 256, it is taken as a single byte character and returned directly. Otherwise, a character name is concocted by concatenating the character set *name* and the hexadecimal character code, with the *prefix* and *suffix* at either end. Character sets in which each ideograph is logically a word will typically use a prefix and suffix of a single blank, while sets in which characters behave like letters will use a void prefix and suffix.

⟨ Class implementations 9 ⟩ +≡
```
string MBCSinterpreter :: getNextDecodedChar(void)
{
  assert(dp ≠ Λ);
  int dc = dp→getNextDecodedChar( );
  if (dc < 0) {
    return "";      /∗ End of input stream ∗/
  }
  if (dc < 256) {
    string r(1, static_cast⟨char⟩(dc));
    return r;
  }
  ostringstream os;
  os.setf(ios :: uppercase);
  os ≪ prefix ≪ name( ) ≪ "-" ≪ hex ≪ dc ≪ dec ≪ suffix;
  return os.str( );
}
```

79. The default implementation of *decodeLine* sets the source to the argument string, then assembles a line by concatenating the results of successive calls to *getNextDecodedChar*.

⟨ Class implementations 9 ⟩ +≡
```
string MBCSinterpreter :: decodeLine(const string &s)
{
  string r = "", t;
  setSource(s);
  while ((t = getNextDecodedChar( )) ≠ "") {
    r += t;
  }
  return r;
}
```

80. GB2312 Interpreter class.
This interpreter class parses `GB2312` ideographs into tokens which downstream parsers can comprehend.

```
⟨ Class definitions 8 ⟩ +≡
  class GB2312_MBCSinterpreter : public MBCSinterpreter {
  public:
    GB2312_MBCSinterpreter()
    {
      setPrefixSuffix("␣","␣");
    }
    virtual string name(void)
    {
      return "GB2312";
    }
  };
```

81. Big5 Interpreter class.
This interpreter class parses `Big5` ideographs into tokens which downstream parsers can comprehend.

```
⟨ Class definitions 8 ⟩ +≡
  class Big5_MBCSinterpreter : public MBCSinterpreter {
  public:
    Big5_MBCSinterpreter()
    {
      setPrefixSuffix("␣","␣");
    }
    virtual string name(void)
    {
      return "Big5";
    }
  };
```

82. Shift-JIS Interpreter class.
This interpreter class parses Shift-JIS ideographs into tokens which downstream parsers can comprehend.

```
⟨ Class definitions 8 ⟩ +≡
  class Shift_JIS_MBCSinterpreter : public MBCSinterpreter {
  public:
    Shift_JIS_MBCSinterpreter()
    {
      setPrefixSuffix("␣","␣");
    }
    virtual string name(void)
    {
      return "Shift_JIS";
    }
    string getNextDecodedChar(void);
  };
```

83. Our *getNextDecodedChar* implementation is a bit more complicated than the default provided by the parent class. In addition to handling ASCII and two byte character codes, we also wish to interpret Katakana single byte characters, which are emitted without spaces between them.

```
⟨ Class implementations 9 ⟩ +≡
  string Shift_JIS_MBCSinterpreter :: getNextDecodedChar(void)
  {
    assert(dp ≠ Λ);
    int dc = dp→getNextDecodedChar( );
    if (dc < 0) {
      return "";      /* End of input stream */
    }
    if (dc < #A1) {
      string r(1, static_cast⟨char⟩(dc));      /* ASCII character */
      return r;
    }
    ostringstream os;
    os.setf(ios :: uppercase);
    if ((dc ≥ #A1) ∧ (dc ≤ #DF)) {
      os ≪ "SJIS-K" ≪ hex ≪ dc ≪ dec;      /* Katakana—don't space around characters */
    }
    else {
      os ≪ prefix ≪ "SJIS-" ≪ hex ≪ dc ≪ dec ≪ suffix;      /* Kanji–space on both sides */
    }
    return os.str( );
  }
```

84. Korean Interpreter class.
This interpreter class parses Korean characters into tokens which downstream parsers can comprehend. This type (usually expressed as a `charset` of `euc-kr`) is uncommon, but we handle it to illustrate an interpreter for an alphabetic non-Western language.

```
⟨ Class definitions 8 ⟩ +≡
  class KR_MBCSinterpreter : public MBCSinterpreter {
  public:
    virtual string name(void)
    {
      return "KR";
    }
  };
```

85. Unicode Interpreter class.

This interpreter class parses Unicode characters into a form which can be comprehended by the parser.

```
⟨ Class definitions 8 ⟩ +≡
  class Unicode_MBCSinterpreter : public MBCSinterpreter {
  public:
    Unicode_MBCSinterpreter( )
    {
      setPrefixSuffix("␣", "␣");
    }
    virtual string name(void)
    {
      return "Unicode";
    }
    string getNextDecodedChar(void);
  };
```

86. Our *getNextDecodedChar* implementation attempts to represent the Unicode characters in a fashion which will best enable the parser to classify them. Characters in the first 256 code positions, which are identical to ISO-8859 are output as ISO characters. Other codes are represented as "UCS-*nnnn*" where *nnnn* is the Unicode code value in hexadecimal. Codes representing iedographs are output separated by spaces while codes for alphanumeric characters are not space-separated.

```
⟨ Class implementations 9 ⟩ +≡
  string Unicode_MBCSinterpreter::getNextDecodedChar(void)
  {
    assert(dp ≠ Λ);
    int dc = dp→getNextDecodedChar( );
    if (dc < 0) {
      return "";     /* End of input stream */
    }
    if (dc ≤ #FF) {
      string r(1, static_cast⟨char⟩(dc));     /* ASCII character */
      return r;
    }
    ostringstream os;
    os.setf(ios::uppercase);
    if (((dc ≥ #3200) ∧ (dc < #D800)) ∨ ((dc ≥ #F900) ∧ (dc < #FAFF))) {
      os ≪ prefix ≪ "UCS-" ≪ hex ≪ dc ≪ dec ≪ suffix;     /* Ideographic–space on both sides */
    }
    else {
      os ≪ "UCS-" ≪ hex ≪ dc ≪ dec;     /* Alphabetic—don't space around characters */
    }
    return os.str( );
  }
```

87. Application string parsers.
An *application string parser* reads files in application-defined formats (for example, word processor documents, spreadsheets, page description languages, etc.) and returns strings included in the file. Unlike *tokenParser* in "byte stream" mode, there is nothing heuristic in the operation of an application string parser—it must understand the structure of the application data file in order to identify and extract strings within it.

The *applicationStringParser* class is the virtual parent of all specific application string parsers. It provides common services to derived classes and defines the external interface. When initialising an *applicationStringParser*, the caller must supply a pointer to the **mailFolder** from which it will be invoked, through which the folder's *nextByte* method will be called to return decoded binary bytes of the application file. It would be *much* cleaner if we could simply supply an arbitrary function which returned the next byte of the stream we're decoding, but that runs afoul of C++'s rules for taking the address of class members. Consequently, we're forced to make *applicationStringParser* co-operate with **mailFolder** to obtain decoded bytes.

```
⟨ Class definitions 8 ⟩ +≡
  class applicationStringParser {
  protected:
  bool error , eof;      /* Error and end of file indicators */
  mailFolder *mf;
  virtual unsigned char get8(void);
  virtual void get8n(unsigned char *buf, const int n)
  {      /* Store next n bytes into buf */
    for (int i = 0; (¬eof) ∧ (i < n); i++) {
      buf[i] = get8( );
    }
  }
  public:
  applicationStringParser(mailFolder *f = Λ) : error (false) , eof(false), mf(Λ)
  {
    setMailFolder(f);
  }
  virtual ~applicationStringParser( )
  { }
  virtual string name(void) const = 0;
  void setMailFolder(mailFolder *f)
  {
    mf = f;
  }
  virtual bool nextString(string &s) = 0;
  virtual void close(void)
  { }
  bool isError(void) const { return error ; }
  bool isEOF(void) const
  {
    return eof;
  }
  bool isOK(void) const
  {
    return (¬isEOF( )) ∧ (¬isError( ));
}
} ;
```

88.

```
⟨ Class implementations 9 ⟩ +≡
unsigned char applicationStringParser :: get8 (void)
{      /* Get next byte, unsigned */
   assert(mf ≠ Λ);
   int ch = mf→nextByte( );
   if (ch ≡ EOF) {
      eof = true;
   }
   return ch & #FF;
}
```

89. Flash stream decoder.

The *flashStream* is a specialisation of **applicationStringParser** which contains all of the logic needed to parse a Macromedia Flash script (`.swf`) file. This class remains abstract in that it does not implement the *nextString* method; that is left for the *flashTextExtractor* class, of which this class is the parent.

This decoder is based on the `swfparse.cpp` program written by David Michie, which is available on the OpenSWF.org site.

```
⟨ Class definitions 8 ⟩ +≡
  class flashStream : public applicationStringParser {
  protected:
    ⟨ Flash file tag values 99 ⟩;
    ⟨ Flash file action codes 100 ⟩;
    ⟨ Flash text field mode definitions 101 ⟩;
    ⟨ Flash file data structures 102 ⟩;    /* Header fields */
    unsigned char sig[3];    /* Signature: "FWS" in ASCII */
    unsigned char version;    /* Version number */
    unsigned int fileLength;    /* Length of entire file in bytes */
    rect frameSize;    /* Frame size in TWIPS */
    unsigned short frameRate;    /* Frames per second (8.8 bit fixed) */
    unsigned short frameCount;    /* Total frames in animation */
      /* Current tag information */
    tagType tType;    /* Tag type */
    unsigned int tDataLen;    /* Length of data chunk */    /* Bit stream decoder storage */
    unsigned int bitBuf, bitPos;
  public:
    flashStream(mailFolder *f = Λ)
    : applicationStringParser(f) { }
    void readHeader(void);    /* Read header into memory */
    void describe(ostream &os = cout);    /* Describe stream */
    bool nextTag(void);    /* Read next tag identifier and length of tag data */
      /* Retrieve properties of current tag */
    tagType getTagType(void) const
    {
      return tType;
    }
    unsigned int getTagDataLength(void) const
    {
      return tDataLen;
    }
    void ignoreTag(unsigned int lookedAhead = 0);
      /* Ignore data for tag we aren't interested in */
  protected:
    ⟨ Read 16 and 32 bit quantities from Flash file 97 ⟩;    /* Skip n bytes of the input stream */
    void skip8n(const int n)
    {
      for (int i = 0; (¬eof) ∧ (i < n); i++) {
        get8();
      }
    }
    void getString(string &s, int n = −1);    /* Bit field decoding methods */
    void initBits(void);
    unsigned int getBits(int n);
    int getSignedBits(const int n);
    void getRect(rect *r);      /* Read a Rectangle specification */
    void getMatrix(matrix *mat);      /* Read a Matrix definition */
  };
```

90. Read the header of the Flash file into memory, validating its signature.

```
⟨ Class implementations 9 ⟩ +≡
  void flashStream::readHeader(void){ sig[0] = get8( );
      sig[1] = get8( );
      sig[2] = get8( ); if (isEOF( ) ∨ (memcmp(sig, "FWS", 3) ≠ 0)) { error = true;
      if (verbose) {
        cerr ≪ "Invalid␣signature␣in␣Flash␣animation␣file." ≪ endl;
      }
      return; } version = get8( );
      fileLength = get32( );
      getRect(&frameSize);
      frameRate = get16( );
      frameCount = get16( ); }
```

91. Write a primate-readable description of the Flash header on the output stream argument *os*, which defaults to *cout*.

```
⟨ Class implementations 9 ⟩ +≡
  void flashStream::describe(ostream &os = cout)
  {
    os ≪ "Flash␣animation␣version␣" ≪ static_cast⟨unsigned int⟩(version) ≪ endl;
    os ≪ "␣␣File␣length:␣" ≪ fileLength ≪ "␣bytes." ≪ endl;
    os ≪ "␣␣Frame␣size:␣␣X:␣" ≪ frameSize.xMin ≪ "␣-␣" ≪ frameSize.xMax ≪ "␣Y:␣" ≪
        frameSize.yMin ≪ "␣-␣" ≪ frameSize.yMax ≪ endl;
    os ≪ "␣␣Frame␣rate:␣" ≪ setprecision(5) ≪ (frameRate/256.0) ≪ "␣fps." ≪ endl;
    os ≪ "␣␣Frame␣count:␣" ≪ frameCount ≪ endl;
  }
```

92. Read the header for the next tag. Each tag begins with a 16 bit field which contains 10 bits of tag identifier and a 6 bit field specifying the number of argument bytes which follow. For tags with arguments of 0 to 62 bytes, the 6 bit field is the data length. For longer tags, the 6 bit length field is set of #3F and a 32 bit quantity giving the tag data length immediately follows. Regardless of the format of the tag header, we store the tag type in *tType* and the number of data bytes in *tDataLen*.

⟨ Class implementations 9 ⟩ +≡

```
bool flashStream::nextTag(void)
{
  unsigned short s = get16();
  unsigned long l;
  if (isOK()) {
    tType = static_cast⟨tagType⟩(s ≫ 6);
    l = s & #3F;
    if (l ≡ #3F) {
      l = get32();      /* Long tag; read 32 bit length */
    }
    if (isOK()) {
      tDataLen = l;
      return tType ≠ stagEnd;
    }
  }      /* In case of error dummy up end tag for sloppy callers */
  tType = stagEnd;
  tDataLen = 0;
  return false;
}
```

93. Having read the tag header, if we decide we aren't interested in the tag, we can simply skip past *tDataLen* argument bytes to advance to the next tag header; *ignoreTag* performs this. If you've read into the tag data before deciding you wish to skip the tag, call *ignoreTag* with the *lookedAhead* argument specifying how many bytes of the tag data you've already read.

⟨ Class implementations 9 ⟩ +≡

```
void flashStream::ignoreTag(unsigned int lookedAhead = 0)
{
  if (isOK()) {
    assert(lookedAhead ≤ tDataLen);
    for (unsigned int i = lookedAhead; i < tDataLen; i++) {
      get8();
    }
  }
}
```

94. Flash files are a little schizophrenic when it comes to the definition of strings. Sometimes they're stored with a leading count byte followed by the given number of bytes of text, while in other places they're stored C style, with a zero terminator byte marking the end of the string. The *getString* method handles both kinds. If called with no length argument, it reads a zero terminated string, otherwise it reads a string of n characters. It's up to the caller to first read the length and pass it as the n argument,

⟨ Class implementations 9 ⟩ +≡

```
void flashStream::getString(string &s, int n = −1)
{
   s = "";
   char ch;
   if (n ≡ −1) {
      while ((ch = get8()) ≠ 0) {
         s += ch;
      }
   }
   else {
      while (n > 0) {
         ch = get8();
         s += ch;
         n−−;
      }
   }
}
```

95. A rectangle is stored as a 5 bit field which specifies the number of bits in the extent fields which follow, which are sign extended when extracted.

⟨ Class implementations 9 ⟩ +≡

```
void flashStream::getRect(rect *r)
{
   initBits();
   int nBits = static_cast⟨int⟩(getBits(5));
   r→xMin = getSignedBits(nBits);
   r→xMax = getSignedBits(nBits);
   r→yMin = getSignedBits(nBits);
   r→yMax = getSignedBits(nBits);
}
```

96. A transformation matrix is stored as separate scale, rotation/skew, and translation terms, each represented as a signed fixed-point value. The scale and rotation/skew terms are optional and are omitted if they are identity—an initial bit indicates whether they are present.

⟨ Class implementations 9 ⟩ +≡

```
void flashStream::getMatrix(matrix * mat)
{
  initBits();      /* Scale terms */
  if (getBits(1)) {
    int nBits = static_cast⟨int⟩(getBits(5));
    mat→a = getSignedBits(nBits);
    mat→d = getSignedBits(nBits);
  }
  else {
    mat→a = mat→d = #00010000L;      /* Identity: omitted */
  }      /* Rotate/skew terms */
  if (getBits(1)) {
    int nBits = static_cast⟨int⟩(getBits(5));
    mat→b = getSignedBits(nBits);
    mat→c = getSignedBits(nBits);
  }
  else {
    mat→b = mat→c = 0;      /* Identity: omitted */
  }      /* Translate terms */
  int nBits = static_cast⟨int⟩(getBits(5));
  mat→tx = getSignedBits(nBits);
  mat→ty = getSignedBits(nBits);
}
```

97. 16 and 32 bit quantities are stored in little-endian byte order. These methods, declared within the class so they're inlined in the interest of efficiency, use the *get8* primitive byte input method to assemble the wider quantities. The *get16n* and *get32n* methods read a series of n consecutive values of the corresponding type into an array.

⟨ Read 16 and 32 bit quantities from Flash file 97 ⟩ ≡

```
unsigned short get16(void)
{
  unsigned short u16;
  u16 = get8();
  u16 |= get8() ≪ 8;
  return u16;
}
unsigned int get32(void)
{
  unsigned int u32;
  u32 = get8();
  u32 |= get8() ≪ 8;
  u32 |= get8() ≪ 16;
  u32 |= get8() ≪ 24;
  return u32;
}
void get16n(unsigned short *buf, const int n)
{
  for (int i = 0; (¬eof) ∧ (i < n); i++) {
    buf[i] = get16();
  }
}
void get32n(unsigned int *buf, const int n)
{
  for (int i = 0; (¬eof) ∧ (i < n); i++) {
    buf[i] = get32();
  }
}
```

This code is used in section 89.

98. Flash files include quantities packed into bit fields, the width of some of which are specified by other fields in the file. The following methods decode these packed fields. Call *initBits* to initialise decoding of a bit field which begins in the next (as yet unread) byte. Then call *getBits* or *getSignedBits* to return an n bit field without or with sign extension respectively.

```
⟨ Class implementations 9 ⟩ +≡
  void flashStream::initBits(void)
  {     /* Reset the bit position and buffer. */
    bitPos = 0;
    bitBuf = 0;
  }     /* Get n bits from the stream. */
  unsigned int flashStream::getBits(int n)
  {
    unsigned int v = 0;
    while (true) {
      int s = n − bitPos;
      if (s > 0) {     /* Consume the entire buffer */
        v |= bitBuf ≪ s;
        n −= bitPos;     /* Get the next buffer */
        bitBuf = get8();
        bitPos = 8;
      }
      else {     /* Consume a portion of the buffer */
        v |= bitBuf ≫ −s;
        bitPos −= n;
        bitBuf &= #FF ≫ (8 − bitPos);     /* mask off the consumed bits */
        return v;
      }
    }
  }     /* Get n bits from the string with sign extension. */
  int flashStream::getSignedBits(const int n)
  {
    signed int v = static_cast⟨int⟩(getBits(n));     /* Is the number negative? */
    if (v & (1 ≪ (n − 1))) {     /* Yes. Extend the sign. */
      v |= −1 ≪ n;
    }
    return v;
  }
```

99. After the header, a Flash file consists of a sequence of *tags*, each of which begins with a 10 bit tag type and a field specifying the number of bytes of tag data which follow. Since each tag specifies its length, unknown tags may be skipped.

```
⟨Flash file tag values 99⟩ ≡    /* Tag values that represent actions or data in a Flash script. */
  typedef enum { stagEnd = 0,    /* End of Flash file—this is always the last tag */
  stagShowFrame = 1,
  stagDefineShape = 2,
  stagFreeCharacter = 3,
  stagPlaceObject = 4,
  stagRemoveObject = 5,
  stagDefineBits = 6,
  stagDefineButton = 7,
  stagJPEGTables = 8,
  stagSetBackgroundColor = 9,
  stagDefineFont = 10,
  stagDefineText = 11,
  stagDoAction = 12,
  stagDefineFontInfo = 13,
  stagDefineSound = 14,    /* Event sound tags. */
  stagStartSound = 15,
  stagDefineButtonSound = 17,
  stagSoundStreamHead = 18,
  stagSoundStreamBlock = 19,
  stagDefineBitsLossless = 20,    /* A bitmap using lossless zlib compression. */
  stagDefineBitsJPEG2 = 21,    /* A bitmap using an internal JPEG compression table. */
  stagDefineShape2 = 22,
  stagDefineButtonCxform = 23,
  stagProtect = 24,    /* This file should not be importable for editing. */
    /* These are the new tags for Flash 3. */
  stagPlaceObject2 = 26,    /* The new style place w/ alpha color transform and name. */
  stagRemoveObject2 = 28,
    /* A more compact remove object that omits the character tag (just depth). */
  stagDefineShape3 = 32,    /* A shape V3 includes alpha values. */
  stagDefineText2 = 33,    /* A text V2 includes alpha values. */
  stagDefineButton2 = 34,    /* A button V2 includes color transform, alpha and multiple actions */
  stagDefineBitsJPEG3 = 35,    /* A JPEG bitmap with alpha info. */
  stagDefineBitsLossless2 = 36,    /* A lossless bitmap with alpha info. */
  stagDefineEditText = 37,    /* An editable Text Field */
  stagDefineSprite = 39,    /* Define a sequence of tags that describe the behavior of a sprite. */
  stagNameCharacter = 40,    /* Name a character definition, character id and a string, (used for
        buttons, bitmaps, sprites and sounds). */
  stagFrameLabel = 43,    /* A string label for the current frame. */
  stagSoundStreamHead2 = 45,    /* For lossless streaming sound, should not have needed this... */
  stagDefineMorphShape = 46,    /* A morph shape definition */
  stagDefineFont2 = 48 ,
  } tagType;
```

This code is used in section 89.

100. Executable actions are encoded in a Flash script as a *stagDoAction* tag, which contains a sequence of action codes, terminated by a zero (*sactionNone*) action. Action codes in the range #00–#7F are single byte codes with no arguments. Action codes from #80 to #FF are followed by a 16 bit field specifying the number of argument bytes which follow. Unknown actions, like tags, may hence be skipped.

⟨ Flash file action codes 100 ⟩ ≡

```
  typedef enum {
    sactionNone = #00,
    sactionNextFrame = #04,
    sactionPrevFrame = #05,
    sactionPlay = #06,
    sactionStop = #07,
    sactionToggleQuality = #08,
    sactionStopSounds = #09,
    sactionAdd = #0A,
    sactionSubtract = #0B,
    sactionMultiply = #0C,
    sactionDivide = #0D,
    sactionEqual = #0E,
    sactionLessThan = #0F,
    sactionLogicalAnd = #10,
    sactionLogicalOr = #11,
    sactionLogicalNot = #12,
    sactionStringEqual = #13,
    sactionStringLength = #14,
    sactionSubString = #15,
    sactionInt = #18,
    sactionEval = #1C,
    sactionSetVariable = #1D,
    sactionSetTargetExpression = #20,
    sactionStringConcat = #21,
    sactionGetProperty = #22,
    sactionSetProperty = #23,
    sactionDuplicateClip = #24,
    sactionRemoveClip = #25,
    sactionTrace = #26,
    sactionStartDragMovie = #27,
    sactionStopDragMovie = #28,
    sactionStringLessThan = #29,
    sactionRandom = #30,
    sactionMBLength = #31,
    sactionOrd = #32,
    sactionChr = #33,
    sactionGetTimer = #34,
    sactionMBSubString = #35,
    sactionMBOrd = #36,
    sactionMBChr = #37,
    sactionHasLength = #80,
    sactionGotoFrame = #81,      /* frame num (WORD) */
    sactionGetURL = #83,      /* url (STR), window (STR) */
    sactionWaitForFrame = #8A,      /* frame needed (WORD), */
      /* actions to skip (BYTE) */
    sactionSetTarget = #8B,     /* name (STR) */
    sactionGotoLabel = #8C,     /* name (STR) */
    sactionWaitForFrameExpression = #8D,     /* frame needed on stack, */
      /* actions to skip (BYTE) */
    sactionPushData = #96,
    sactionBranchAlways = #99,
    sactionGetURL2 = #9A,
    sactionBranchIfTrue = #9D,
    sactionCallFrame = #9E,
    sactionGotoExpression = #9F
  } actionCode;
```

This code is used in section 89.

101. Here we define the various mode bits which occur in font and text related tags. Many of these bits are irrelevant to our mission of string parsing, but we define them all anyway.

⟨ Flash text field mode definitions 101 ⟩ ≡

```
  typedef enum {     /* Flag bits for DefineFontInfo */
    fontUnicode = #20,
    fontShiftJIS = #10,
    fontANSI = #08,
    fontItalic = #04,
    fontBold = #02,
    fontWideCodes = #01
  } fontFlags;
  typedef enum {     /* Flag bits for text record type 1 */
    isTextControl = #80,
    textHasFont = #08,
    textHasColor = #04,
    textHasYOffset = #02,
    textHasXOffset = #01
  } textFlags;
  typedef enum {     /* Flag bits for DefineEditText */
    seditTextFlagsHasFont = #0001,
    seditTextFlagsHasMaxLength = #0002,
    seditTextFlagsHasTextColor = #0004,
    seditTextFlagsReadOnly = #0008,
    seditTextFlagsPassword = #0010,
    seditTextFlagsMultiline = #0020,
    seditTextFlagsWordWrap = #0040,
    seditTextFlagsHasText = #0080,
    seditTextFlagsUseOutlines = #0100,
    seditTextFlagsBorder = #0800,
    seditTextFlagsNoSelect = #1000,
    seditTextFlagsHasLayout = #2000
  } editTextFlags;
```

This code is used in section 89.

102. The following data structures are used to represent rectangles and transformation matrices. We don't do anything with these quantities, but we need to understand their structure in order to skip over them while looking for fields we are interested in.

```
⟨ Flash file data structures 102 ⟩ ≡
  typedef struct {
    int xMin, xMax, yMin, yMax;
  } rect;
  typedef struct {
    int a;
    int b;
    int c;
    int d;
    int tx;
    int ty;
  } matrix;
```

This code is used in section 89.

103. Flash text extractor.

The *flashTextExtractor* extends **flashStream** to parse tags containing text fields and return them with the *nextString* method. We define this as a separate class in order to encapsulate all of the string parsing machinery in one place, while leaving **flashStream** a general-purpose `.swf` file parser adaptable to other purposes.

```
⟨ Class definitions 8 ⟩ +≡
  class flashTextExtractor : public flashStream {
  protected: map⟨unsigned short, vector⟨unsigned short⟩ ∗⟩ fontMap;
    map⟨unsigned short, unsigned short⟩ fontGlyphCount;
    map⟨unsigned short, fontFlags⟩ fontInfoBits;
    queue⟨string⟩ strings;
    bool initialised;      /∗ Options ∗/
    bool textOnly;      /∗ Return only text (not font names, URLs, etc.) ∗/
  public: flashTextExtractor(mailFolder ∗f = Λ)
    : flashStream(f), initialised(false), textOnly(false) { }
    ~flashTextExtractor( )
    {
      close( );
    }
    virtual string name(void) const
    {
      return "Flash";
    }
    void setTextOnly(const bool tf)
    {
      textOnly = tf;
    }
    bool getTextOnly(void) const
    {
      return textOnly;
    }
    bool nextString(string &s);      /∗ Return next string from Flash file ∗/
    virtual void close(void)
    {
      while (¬fontMap.empty( )) {
        delete fontMap.begin( )→second;
        fontMap.erase(fontMap.begin( ));
      }
      fontGlyphCount.clear( );
      fontInfoBits.clear( );
    }
  };
```

104. Return the next string (which may contain any number of tokens) from the Flash file. If the *strings* queue contains already-parsed strings, return and delete the the item at the head of the queue. Otherwise, we parse our way through the Flash file, adding any strings which appear in tags to the *strings* queue. If, after parsing a tag, we find *strings* non-empty, we return the first item in the queue. The method returns *true* if a string was stored and *false* when the end of the Flash file is encountered.

The first time this method is called, we read the Flash file header and validate it. If an error occurs in the process, we treat the event as a logical end of file.

```
⟨ Class implementations 9 ⟩ +≡
  bool flashTextExtractor::nextString(string &s)
  {
    if (¬initialised) {
      initialised = true;
      readHeader();
      if (¬isOK()) {
        if (verbose) {
          cerr ≪ "Invalid␣header␣in␣Flash␣application␣file." ≪ endl;
          close();
          while (¬isEOF()) {
            get8();      /* Discard contents after error */
          }
          return false;
        }
      }
    }
    while (true) {
    haveStrings:
      ⟨ Check for strings in the queue and return first if queue not empty 105 ⟩;
      while ((¬isEOF()) ∧ (¬isError()) ∧ nextTag()) {
        unsigned int variant = 0;      /* Twiddley-puke variant type for tags */
        switch (tType) {
        case stagDefineFont:
          ⟨ Parse Flash DefineFont tag 106 ⟩;
          break;
        case stagDefineFont2:
          ⟨ Parse Flash DefineFont2 tag 107 ⟩;
          break;
        case stagDefineFontInfo:
          ⟨ Parse Flash DefineFontInfo tag 108 ⟩;
          break;
        case stagDefineText2:      /* Like stagDefineText, but colour is RGBA */
          variant = 2;      /* Note fall-through */
        case stagDefineText:
          ⟨ Parse Flash DefineText tags 109 ⟩;
          break;
        case stagDefineEditText:
          ⟨ Parse Flash DefineEditText tag 111 ⟩;
          break;
        case stagFrameLabel:
          ⟨ Parse Flash FrameLabel tag 112 ⟩;
          break;
        case stagDoAction:
          ⟨ Parse Flash DoAction tag 113 ⟩;
          break;
        default:
#ifdef FLASH_PARSE_DEBUG
          cout ≪ "nextString␣ignoring␣tag␣type␣" ≪ getTagType( ) ≪ "␣data␣length:␣" ≪
              getTagDataLength( ) ≪ endl;
#endif
          ignoreTag( );
          break;
        }
        if (¬strings.empty( )) {
          goto haveStrings;
        }
      }
      if (strings.empty( )) {
        break;
      }
    }
    return false;
  }
```

105. Since a single tag may contain any number of strings, we place strings extracted from a tag in the *strings* queue. Then, after we're done digesting the tag, if the queue is non-empty, we return the first string from it. Subsequent calls return strings from the queue until it's empty, at which time we resume scouring the Flash file for more strings.

⟨ Check for strings in the queue and return first if queue not empty 105 ⟩ ≡

```
  if (¬strings.empty( )) {
    s = strings.front( );
    strings.pop( );
    return true;
  }
```

This code is used in section 104.

106. The DefineFont tag actually contains only one thing of interest to us: the number of glyphs in the font. We save the glyph count in the *fontFlyphCount* map, tagged by the font ID.

⟨ Parse Flash DefineFont tag 106 ⟩ ≡

```
  {
#ifdef FLASH_PARSE_DEBUG
    cout ≪ "DefineFont" ≪ endl;
#endif
    unsigned short fontID = get16( );
    unsigned int offsetTable = get16( );
#ifdef FLASH_PARSE_DEBUG
    cout ≪ "␣␣Font␣ID:␣" ≪ fontID ≪ endl;
    cout ≪ "␣␣Glyph␣count:␣" ≪ (offsetTable/2) ≪ endl;
#endif
    fontGlyphCount.insert(make_pair(fontID, offsetTable/2));
    ignoreTag(2 ∗ 2);
  }
```

This code is used in section 104.

107. The DefineFont2 tag adds a font name to the fields in the original DefineFont tag. We consider this font name as an eligible string if the *textOnly* constraint isn't *true*.

```
⟨ Parse Flash DefineFont2 tag 107 ⟩ ≡
  {
#ifdef FLASH_PARSE_DEBUG
    cout ≪ "DefineFont2" ≪ endl;
#endif
    unsigned short fontID = get16( );
    get16( );    /* Flag bits */    /* Parse the font name */
    unsigned int fontNameLen = get8( );
    string fontName;
    getString(fontName, fontNameLen);
    if (¬textOnly) {
      strings.push(fontName);
    }    /* Get the number of glyphs. */
    unsigned int nGlyphs = get16( );
    fontGlyphCount.insert(make_pair(fontID, nGlyphs));
    ignoreTag(2 + 2 + 1 + fontNameLen + 2);
  }
```

This code is used in section 104.

108. The DefineFontInfo tag is crucial to decoding Flash text strings. Text in Flash files is stored a glyph indices within a font. The font can, in the general case, be defined by an arbitrary stroked path outline, independent of any standard character set. For fonts which employ standard character sets, the optional DefineFontInfo identifies the character set and provides the mapping from the glyph indices to characters in the font's character set. We save these in maps indexed by the font ID so we can look them up when we encounter text in that font.

```
⟨ Parse Flash DefineFontInfo tag 108 ⟩ ≡
  {
#ifdef FLASH_PARSE_DEBUG
    cout ≪ "DefineFontInfo" ≪ endl;
#endif
    unsigned short fontID = get16( );
    unsigned int fontNameLen = get8( );
    string fontName;
    getString(fontName, fontNameLen);
    if (¬textOnly) {
      strings.push(fontName);
    }
    fontFlags fFlags = static_cast⟨fontFlags⟩(get8( ));
    map⟨unsigned short, unsigned short⟩ :: iterator fp = fontGlyphCount.find(fontID);
    if (fp ≡ fontGlyphCount.end( )) {
      if (verbose) {
        cerr ≪ "DefineFontInfo␣for␣font␣ID␣" ≪ fontID ≪
            "␣without␣previous␣DefineFont." ≪ endl;
      }
      ignoreTag(4);
    }
    else {
      unsigned nGlyphs = fp→second;
      vector⟨unsigned short⟩ *v = new vector⟨unsigned short⟩(nGlyphs);
      fontMap.insert(make_pair(fontID, v));
      fontInfoBits.insert(make_pair(fontID, fFlags));
      for (unsigned int g = 0; g < nGlyphs; g++) {
        if (fFlags & fontWideCodes) {
          (*v)[g] = get16( );
        }
        else {
          (*v)[g] = get8( );
        }
      }
    }
  }
```

This code is used in section 104.

109. Most of the text we're really interested in will be found in the DefineText tag and its younger sibling DefineText2. After spitting out the various wobbly green parts, we digest the list of glyphs composing the text, going back to the font definition to claw them back into civilised language which we can filter.

```
⟨ Parse Flash DefineText tags 109 ⟩ ≡
  {
#ifdef FLASH_PARSE_DEBUG
    unsigned short textID = get16();
    cout ≪ "DefineText.␣␣ID␣=␣" ≪ textID ≪ endl;
#else
    get16();    /* Ignore textID */
#endif
    rect tr;
    getRect(&tr);
    matrix tm;
    getMatrix(&tm);
    unsigned short textGlyphBits = get8();
    unsigned short textAdvanceBits = get8();
    int fontId = −1;
    map⟨unsigned short, vector⟨unsigned short⟩ *⟩::iterator fontp = fontMap.end();
    map⟨unsigned short, unsigned short⟩::iterator fgcp = fontGlyphCount.end();
    unsigned int fGlyphs = 0;
    fontFlags fFlags = static_cast⟨fontFlags⟩(0);
    vector⟨unsigned short⟩ *fontChars = Λ;    /* Now it's a matter of parsing the text records */
    while (true) {
      unsigned int textRecordType = get8();
      if (textRecordType ≡ 0) {
        break;    /* 0 indicates end of text records */
      }
      if (textRecordType & isTextControl) {
#ifdef FLASH_PARSE_DEBUG
        cout ≪ "Text␣control␣record." ≪ endl;
#endif
        if (textRecordType & textHasFont) {
          fontId = get16();
#ifdef FLASH_PARSE_DEBUG
          cout ≪ "␣␣␣␣fontId:␣" ≪ fontId ≪ endl;
#endif
          fgcp = fontGlyphCount.find(fontId);
          if (fgcp ≡ fontGlyphCount.end()) {
            fontp = fontMap.end();
            if (verbose) {
              cerr ≪ "Flash␣DefineText␣item␣references␣undefined␣font␣ID␣" ≪ fontId ≪
                  endl;
            }
          }
          else {
            fGlyphs = fgcp→second;
            fontChars = fontMap.find(fontId)→second;
```

```
                fFlags = fontInfoBits.find(fontId)→second;
              }
            }
            if (textRecordType & textHasColor) {
#ifdef FLASH_PARSE_DEBUG
              int r = get8();
              int g = get8();
              int b = get8();
              if (variant ≡ 2) {
                int a = get8();      /* Alpha (transparency) channel */
                cout ≪ "␣␣␣␣tfontColour:␣(" ≪ r ≪ "," ≪ g ≪ "," ≪ b ≪ "," ≪ a ≪ ")" ≪ endl;
              }
              else {
                cout ≪ "␣␣␣␣tfontColour:␣(" ≪ r ≪ "," ≪ g ≪ "," ≪ b ≪ ")" ≪ endl;
              }
#else
              skip8n(3);      /* Skip R, G, B bytes */
#endif
            }
            if (textRecordType & textHasXOffset) {
#ifdef FLASH_PARSE_DEBUG
              int iXOffset = get16();
              cout ≪ "␣␣␣␣X␣offset␣" ≪ iXOffset ≪ endl;
#else
              get16();      /* Skip text X offset */
#endif
            }
            if (textRecordType & textHasYOffset) {
#ifdef FLASH_PARSE_DEBUG
              int iYOffset = get16();
              cout ≪ "␣␣␣␣Y␣offset␣" ≪ iYOffset ≪ endl;
#else
              get16();      /* Skip text Y offset */
#endif
            }
            if (textRecordType & textHasFont) {
#ifdef FLASH_PARSE_DEBUG
              int iFontHeight = get16();
              cout ≪ "␣␣␣␣Font␣Height:␣" ≪ iFontHeight ≪ endl;
#else
              get16();      /* Skip text font height */
#endif
            }
          }
          else {      /* Type 0: Glyph record */
#ifdef FLASH_PARSE_DEBUG
            cout ≪ "Text␣glyph␣record." ≪ endl;
#endif
            unsigned int nGlyphs = textRecordType & #7F;
            initBits();
            string s = "";
            for (unsigned int i = 0; i < nGlyphs; i++) {
               unsigned int iIndex = getBits(textGlyphBits);
#ifdef FLASH_PARSE_DEBUG
               unsigned int iAdvance = getBits(textAdvanceBits);
               cout ≪ "[" ≪ iIndex ≪ "," ≪ iAdvance ≪ "]␣" ≪ flush;
#else
               getBits(textAdvanceBits);     /* Ignore text advance distance */
#endif
               if (fontId < 0) {
                  if (verbose) {
                     cerr ≪ "Flash␣DefineText␣does␣not␣specify␣font." ≪ endl;
                  }
               }
               else if (fgcp ≠ fontGlyphCount.end()) {
                  if (iIndex ≥ fGlyphs) {
                     if (verbose) {
                        cerr ≪ "Flash␣DefineText␣glyph␣index␣" ≪ iIndex ≪
                             "␣exceeds␣font␣size␣of␣" ≪ fGlyphs ≪ "." ≪ endl;
                     }
                  }
                  else {
                     if (fFlags & fontWideCodes) {
                        unsigned int wc = (*fontChars)[iIndex];
                        s += static_cast⟨char⟩((wc ≫ 8) & #FF);
                        s += static_cast⟨char⟩(wc & #FF);
                     }
                     else {
                        s += static_cast⟨char⟩((*fontChars)[iIndex]);
                     }
                  }
               }
            }
#ifdef FLASH_PARSE_DEBUG
            cout ≪ endl;
            cout ≪ "Decoded:␣(" ≪ s ≪ ")" ≪ endl;
#endif
            ⟨ Decode non-ANSI Flash text 110 ⟩;
            strings.push(s);
         }
      }
   }
```

This code is used in section 104.

110. Text strings in a Flash file can be encoded in Shift-JIS and Unicode in addition to ANSI characters. If the font if flagged as using one of those encodings, decode it into an ANSI representation.

```
⟨ Decode non-ANSI Flash text 110 ⟩ ≡
  if (fFlags & fontUnicode) {
    UCS_2_Unicode_MBCSdecoder mbd_ucs;      /* Unicode decoder */
    Unicode_MBCSinterpreter mbi_ucs;      /* Unicode interpreter */

    mbi_ucs.setDecoder(mbd_ucs);
    s = mbi_ucs.decodeLine(s);
  }
  else if (fFlags & fontShiftJIS) {
    Shift_JIS_MBCSdecoder mbd_sjis;      /* Shift-JIS decoder */
    Shift_JIS_MBCSinterpreter mbi_sjis;      /* Shift-JIS interpreter */

    mbi_sjis.setDecoder(mbd_sjis);
    s = mbi_sjis.decodeLine(s);
  }
```

This code is used in section 109.

111. Of course, there isn't just text, there's *editable text*, where morons can type in their credit card numbers after receiving "so cool a Flash". We deem any initial text in the edit field a string, as well as the variable name, unless *textOnly* is *true*.

```
⟨ Parse Flash DefineEditText tag 111 ⟩ ≡
  {
#ifdef FLASH_PARSE_DEBUG
    cout ≪ "Edit␣text␣record." ≪ endl;
#endif
    get16();
    rect rBounds;
    getRect(&rBounds);
    unsigned int flags = get16();
#ifdef FLASH_PARSE_DEBUG
    cout ≪ "DefineEditText.␣␣Flags␣=␣0x" ≪ hex ≪ flags ≪ dec ≪ endl;
#endif
    if (flags & seditTextFlagsHasFont) {
#ifdef FLASH_PARSE_DEBUG
      unsigned short uFontId = get16();
      unsigned short uFontHeight = get16();
      cout ≪ "FontId:␣" ≪ uFontId ≪ "␣␣FontHeight:␣" ≪ uFontHeight ≪ endl;
#else
      get16();
      get16();
#endif
    }
    if (flags & seditTextFlagsHasTextColor) {
      skip8n(4);    /* Skip colour (including alpha transparency) */
    }
    if (flags & seditTextFlagsHasMaxLength) {
#ifdef FLASH_PARSE_DEBUG
      int iMaxLength = get16();
      printf("length:%d␣", iMaxLength);
#else
      get16();
#endif
    }
    if (flags & seditTextFlagsHasLayout) {
      skip8n(1 + (2 * 4));
    }
    string varname;
    getString(varname);
    if (¬textOnly) {
      strings.push(varname);    /* Emit variable name as a string */
    }
    if (flags & seditTextFlagsHasText) {
      string s;
      char c;
      while ((c = get8()) ≠ 0) {
        s += c;
      }
```

```
        strings.push(s);
      }
    }
```

This code is used in section 104.

112. Frames in Flash files can have labels, which can be used to jump to them. If *textOnly* is not set, we parse these labels and return them as strings, since they will frequently identify Flash files which appear in junk mail.

```
⟨ Parse Flash FrameLabel tag 112 ⟩ ≡
  {
    string s;
    getString(s);
    if (¬textOnly) {
      strings.push(s);
    }
  }
```

This code is used in section 104.

113. Some of the DoAction tags contain string we might be interested in perusing. Walk through the action items in a DoAction tag and push any relevant strings onto the *strings* queue.

```
⟨ Parse Flash DoAction tag 113 ⟩ ≡
  {
#ifdef FLASH_PARSE_DEBUG
    cout ≪ "Do␣action:" ≪ endl;
#endif
    actionCode ac;
    while (isOK() ∧ (ac = static_cast⟨actionCode⟩(get8())) ≠ sactionNone) {
      unsigned int dlen = 0;
      if ((ac & #80) ≠ 0) {
        dlen = get16();
      }
      switch (ac) {
      case sactionGetURL:
        {
          string url, target;
          getString(url);
          getString(target);
          if (¬textOnly) {
            strings.push(url);
          }
          strings.push(target);
        }
        break;
      default:
        if (dlen > 0) {
          skip8n(dlen);
        }
#ifdef FLASH_PARSE_DEBUG
        cout ≪ "␣␣Skipping␣action␣code␣0x" ≪ hex ≪ ac ≪ dec ≪ "␣data␣length␣" ≪ dlen ≪
            endl;
#endif
        break;
      }
    }
  }
```

This code is used in section 104.

114. PDF text extractor.

The *pdfTextExtractor* decodes Portable Document File `.pdf` files by opening a pipe to the `pdftotext` program. Since this program cannot read a PDF document from standard input, we transcribe the PDF stream to a temporary file which is passed to `pdftotext` on the command line; the extracted text is directed to standard output whence it can be read through the pipe. The temporary file is deleted after the PDF decoding is complete. Natually, this facility is available only if the system provides `pdftotext` and the machinery needed to connect to it.

⟨ Class definitions 8 ⟩ +≡

```
#ifdef HAVE_PDF_DECODER
  class pdfTextExtractor : public applicationStringParser {
  protected: bool initialised;
    ifstream is;
    FILE *ip;
#ifdef HAVE_MKSTEMP
    char tempfn[256];
#else
    char tempfn[L_tmpnam + 2];
#endif
  public: pdfTextExtractor(mailFolder *f = Λ)
    : applicationStringParser(f), initialised(false), ip(Λ) { }
    ~pdfTextExtractor( )
    {
      close( );
    }
    virtual string name(void) const
    {
      return "PDF";
    }
    bool nextString(string &s);
    virtual void close(void)
    {
      if (ip ≠ Λ) {
        pclose(ip);
        remove(tempfn);
        ip = Λ;
      }
    }
  };
#endif
```

115. Since `pdftotext` cannot read a PDF file from standard input, we're forced to transcribe the content to a temporary file. We do this the first time *nextString* is called, setting the *initialised* flag once the deed is done. Subsequent calls simply return the decoded text from the pipe, closing things down when end of file is encountered.

```
⟨ Class implementations 9 ⟩ +≡
#ifdef HAVE_PDF_DECODER
  bool pdfTextExtractor::nextString(string &s)
  {
    if (¬initialised) {
      initialised = true;
      ⟨ Transcribe PDF document to temporary file 116 ⟩;
      ⟨ Create pipe to pdftotext decoder 117 ⟩;
    }
    if (ip ≡ Λ) {
      return false;      /* Could not open pipe; fake EOF */
    }
    if (getline(is, s) ≠ Λ) {
      return true;
    }
    close();
    return false;
  }
#endif
```

116. Read the PDF document text and export to a temporary file whence `pdftotext` can read it. We generate a unique name for the temporary file with *mkstemp* or, if the system doesn't provide that function, the POSIX *tmpnam* alternative.

```
⟨ Transcribe PDF document to temporary file 116 ⟩ ≡
#ifdef HAVE_MKSTEMP
  strcpy(tempfn, "PDF_decode_XXXXXX");
  mkstemp(tempfn);
#else
  tmpnam(tempfn);
#endif
  ofstream pdfstr(tempfn, ios::out | ios::binary); if (¬pdfstr) {
      cerr ≪ "Cannot␣create␣PDF␣temporary␣file␣" ≪ tempfn ≪ endl; error =
      eof = true;
  return false; }
  while (¬isEOF()) {
    pdfstr ≪ get8();
  }
  pdfstr.close();
```

This code is used in section 115.

117. Since `pdftotext` does all the heavy lifting here, we need only invoke it with *popen*, which is bound to the C++ input stream we use to read the decoded text.

```
⟨ Create pipe to pdftotext decoder 117 ⟩ ≡
  string pdfcmd = "pdftotext␣";
  pdfcmd += tempfn;
  pdfcmd += "␣-";
  ip = popen(pdfcmd.c_str(), "r"); if (ip ≡ Λ) { cerr ≪ "Cannot␣open␣pipe␣to␣pdftotext." ≪ endl;
      error = eof = true;
  return false; } is.attach(fileno(ip));
```

This code is used in section 115.

118. Mail folder.
The **mailFolder** class returns successive lines from a mail folder bound to an input stream.

```
⟨ Class definitions 8 ⟩ +≡
  ⟨ Configure compression suffix and command 121 ⟩
      class mailFolder {
      public:
        istream *is;    /* Stream to read mail folder from */
        dictionaryWord::mailCategory category;    /* Category (Mail or Junk) */
        unsigned int nLines;    /* Number of lines in folder */
        unsigned int nMessages;    /* Number of messages read so far */
        bool newMessage;    /* On first line of new message ? */
        bool inHeader;    /* Within message header section */
        string lookAheadLine;    /* Line to save look ahead while parsing headers */
        bool lookedAhead;    /* Have we a look ahead line ? */
        ifstream isc;    /* Input stream for (possibly compressed) input file */
          /* Compressed file decoding */
#ifdef COMPRESSED_FILES
        FILE *ip;    /* File handle used for popen pile to decompressor */
#endif
#ifdef HAVE_DIRECTORY_TRAVERSAL    /* Directory traversal */
        bool dirFolder;    /* Are we reading a directory folder ? */
        DIR * dh;    /* Handle for readdir */
        string dirName, cfName;    /* Directory name and current file name in directory */
        string pathSeparator;    /* System path separator */
        ifstream ifdir;    /* Stream to read file in directory */
        istringstream nullstream;    /* Null stream for empty directory case */
#endif    /* Body encoding properties */
        string bodyContentType;    /* Content-Type */
        string bodyContentTypeCharset;    /*     charset= */
        string bodyContentTransferEncoding;    /* Content-Transfer-Encoding */
        string partBoundary;    /* Mime part boundary sentinel */
        bool multiPart;    /* Is message MIME multi-part ? */
        bool inPartHeader;    /* In MIME part header ? */
        unsigned int partHeaderLines;    /* Number of lines in part header */
        stack⟨string⟩ partBoundaryStack;
          /* stack of part boundaries for multipart/alternative nesting */
          /* MIME properties of current part */
        string mimeContentType;    /* Content-Type */
        string mimeContentTypeCharset;    /*     charset= */
        string mimeContentTypeBoundary;    /*     boundary= */
        string mimeContentTransferEncoding;    /* Content-Transfer-Encoding */
          /* MIME decoders */
        MIMEdecoder *mdp;    /* Active MIME decoder if any */
        identityMIMEdecoder imd;    /* Identity MIME decoder for testing */
        base64MIMEdecoder bmd;    /* Base64 MIME decoder for testing */
        sinkMIMEdecoder smd;    /* Sink MIME decoder */
        quotedPrintableMIMEdecoder qmd;    /* Quoted-Printable MIME decoder */
          /* Multi-byte character set decoding */
        MBCSinterpreter *mbi;    /* Active multi-byte character set interpreter or Λ */
        EUC_MBCSdecoder mbd_euc;    /* EUC decoder */
        GB2312_MBCSinterpreter mbi_gb2312;    /* GB2312 interpreter */
        Big5_MBCSdecoder mbd_big5;    /* Big5 decoder */
        Big5_MBCSinterpreter mbi_big5;    /* Big5 interpreter */
        KR_MBCSinterpreter mbi_kr;    /* Korean (euc-kr) interpreter */
        UTF_8_Unicode_MBCSdecoder mbd_utf_8;    /* Unicode UTF-8 decoder */
        Unicode_MBCSinterpreter mbi_unicode;    /* Unicode interpreter */
          /* Application file string parsing */
        applicationStringParser *asp;    /* Application string parser or NULL if none */
        flashTextExtractor aspFlash;    /* Flash animation string parser */
#ifdef HAVE_PDF_DECODER
        pdfTextExtractor aspPdf;    /* PDF string parser */
#endif    /* Byte stream decoding */
        bool byteStream;    /* Extract probable strings from binary files ? */
        list⟨string⟩ *tlist;    /* Message transcript list */
        list⟨string⟩ *dlist;    /* Diagnostic message contents list */
        mailFolder(istream &i,
                dictionaryWord::mailCategory cat = dictionaryWord::Unknown)
        {
#ifdef COMPRESSED_FILES
          ip = Λ;
#endif
          set(&i, cat);
        }
        mailFolder(string fname,
                dictionaryWord::mailCategory cat = dictionaryWord::Unknown)
        {
#ifdef COMPRESSED_FILES
          ip = Λ;
#endif
          ⟨Check whether folder is a directory of messages 124⟩;
#ifdef HAVE_DIRECTORY_TRAVERSAL
          if (¬dirFolder) {
#endif
#ifdef COMPRESSED_FILES
            ⟨Check for symbolic link to compressed file 122⟩;
            if (jname.rfind(Compressed_file_type) ≠ string::npos) {
              ⟨Open pipe to read compressed file 123⟩;
            }
            else {
#endif
              if (fname ≡ "-") {
                is = &cin;
              }
              else {
                isc.open(fname.c_str());
                is = &isc;
              }
#ifdef COMPRESSED_FILES
            }
#endif
#ifdef HAVE_DIRECTORY_TRAVERSAL
          }
#endif
```

```
            if (¬(∗is)) {
              cerr ≪ "Cannot␣open␣mail␣folder␣file␣" ≪ fname ≪ endl;
              exit(1);
            }
            set(is, cat);
          }
          ~mailFolder( )
          {
#ifdef COMPRESSED_FILES
            if (ip ≠ Λ) {
              pclose(ip);
            }
#endif
          }
          void set(istream ∗i, dictionaryWord :: mailCategory cat = dictionaryWord :: Unknown)
          {
            is = i;
            nLines = nMessages = 0;
            lookedAhead = false;
            lookAheadLine = "";
            category = cat;
            dlist = Λ;
            tlist = Λ;
            ⟨ Reset MIME decoder state 131 ⟩;
            bodyContentType = bodyContentTypeCharset = bodyContentTransferEncoding = "";
          }
          void setCategory(dictionaryWord :: mailCategory c)
          {
            category = c;
          }
          dictionaryWord :: mailCategory getCategory(void) const
          {
            return category;
          }
          bool nextLine(string &s);
          int nextByte(void);
#ifdef HAVE_DIRECTORY_TRAVERSAL
          bool findNextFileInDirectory(string &fname);
          bool openNextFileInDirectory(void);
#endif
          static void stringCanonicalise(string &s);
          static bool compareHeaderField(string &s, const string target, string &arg);
          static bool parseHeaderArgument(string &s, const string target, string &arg);
          bool isNewMessage(void) const
          {
            return newMessage;
          }
          unsigned int getMessageCount(void) const
          {
            return nMessages;
```

```
  }
  unsigned int getLineCount(void) const
  {
    return nLines;
  }
  bool isByteStream(void) const
  {
    return byteStream;
  }
  void describe(ostream &os = cout) const
  {
    os ≪ "Mail␣folder.␣␣Category:␣" ≪ dictionaryWord::categoryName(category) ≪
        endl;
    os ≪ "␣␣Lines:␣" ≪ getLineCount() ≪ "␣␣Messages:␣" ≪ getMessageCount() ≪ endl;
  }
  void setDiagnosticList(list⟨string⟩ *lp)
  {
    dlist = lp;
  }
  void setTranscriptList(list⟨string⟩ *lp)
  {
    tlist = lp;
  }
  void writeMessageTranscript(ostream &os = cout);
  void writeMessageTranscript(const string fname = "-");
  void reportParserDiagnostic(const string s) const;
  void reportParserDiagnostic(const ostringstream &os) const;
};
```

119. The *nextLine* method returns the next line from the mail folder to the caller, while parsing the mail folder into headers, recognising MIME multi-part messages and their boundaries and encodings. We wrap a grand **while** loop around the entire function so code within it can ignore the current input line (which may, depending on where you are in the process, be the concatenation of header lines with continuations), with a simple **continue**.

```
⟨ Class implementations 9 ⟩ +≡
  bool mailFolder::nextLine(string &s)
  {
    while (true) {
      bool decoderEOF = false;
      if (lookedAhead) {
        s = lookAheadLine;
        lookedAhead = false;
      }
      else {
        if (mdp ≠ Λ) {
          if ((asp ≠ Λ) ? (¬asp→nextString(s)) : (¬(mdp→getDecodedLine(s)))) {
            if (asp ≠ Λ) {
              asp→close();
              asp = Λ;
            }
            s = mdp→getTerminatorSentinel();
            decoderEOF = mdp→isEndOfFile();
            if (decoderEOF) {
              s = "";
            }
            if (Annotate('d')) {
              ostringstream os;
              os ≪ "Closing␣out␣" ≪ mdp→name() ≪ "␣decoder.␣␣" ≪
                  mdp→getEncodedLineCount() ≪ "␣lines␣decoded.";
              reportParserDiagnostic(os);
              os.str("");
              os ≪ "End␣sentinel:␣" ≪ s;
              reportParserDiagnostic(os);
            }
            ⟨ Reset MIME decoder state 131 ⟩;
            inPartHeader = ¬((s.substr(0, 2) ≡ "--") ∧ (s.substr(2,
                partBoundary.length()) ≡ partBoundary) ∧ (s.substr(partBoundary.length() + 2,
                2) ≡ "--"));
            if ((¬inPartHeader) ∧ (¬(partBoundaryStack.empty()))) {
              partBoundary = partBoundaryStack.top();
              partBoundaryStack.pop();
            }
          }
        }
        else {
          if (¬getline(*is, s)) {
            ⟨ Advance to next file if traversing directory 127 ⟩;
            return false;
          }
        }
      }
      $nLines$++;
      if (($mdp \equiv \Lambda) \land (tlist \neq \Lambda) \land (\neg decoderEOF$)) {
        $tlist \rightarrow push\_back(s)$;
      }
      ⟨ Check for start of new message in folder 128 ⟩;
      ⟨ Eliminate any trailing space from line 129 ⟩;
      ⟨ Process message header lines 130 ⟩;
      ⟨ Parse MIME part header 152 ⟩;
      ⟨ Check for MIME part sentinel 138 ⟩;
      ⟨ Decode multiple byte character set 139 ⟩;
      return true;
    }
  }
```

120. The *nextByte* method is used by the *tokenParser* when scouring byte stream data for plausible strings. It must only be used when *byteStream* is set. It returns the next byte from the stream or −1 at the end of the stream and cancels *byteStream* mode. How we get out of here depends on a fairly intimate mutual understanding between **mailFolder** and *tokenParser* of each other's innards.

⟨ Class implementations 9 ⟩ +≡

```
  int mailFolder :: nextByte(void)
  {
    assert($mdp \neq \Lambda$);
    int $c = mdp \rightarrow getDecodedChar()$;
    if ($c < 0$) {
      $byteStream = false$;
      if (Annotate('d')) {
        ostringstream os;
        os ≪ "End␣of␣byte␣stream.␣␣Deactivating␣byte␣stream␣parser.";
        reportParserDiagnostic(os);
      }
    }
    return c;
  }
```

121. The type of compression and command required to expand compressed files may differ from system to system. The following code, conditional based on variables determined by the `autoconf` process, defines the file suffix denoting a compressed file and the corresponding command used to decode it. We only support one type of compression on a given system; if `gzip` is available, we use it in preference to `compress`.

```
⟨ Configure compression suffix and command 121 ⟩ ≡
#ifdef HAVE_POPEN
#if (defined HAVE_GUNZIP) ∨ (defined HAVE_GZCAT) ∨ (defined HAVE_GZIP)
  # define COMPRESSED_FILES
      static const char Compressed_file_type[] = ".gz";
      static const char Uncompress_command[] =
      # if (defined HAVE_GUNZIP)
      "gunzip␣-c"
      # elif (defined HAVE_GZCAT)
      "gzcat"
      # elif (defined HAVE_GZIP)
      "gzip␣-cd"
      # endif
      ;
#elif (defined HAVE_ZCAT) ∨ (defined HAVE_UNCOMPRESS) ∨ (defined HAVE_COMPRESS)
      # define COMPRESSED_FILES
          static const char Compressed_file_type[] = ".Z";
          static const char Uncompress_command[] =
          # if (defined HAVE_ZCAT)
          "zcat"
          # elif (defined HAVE_UNCOMPRESS)
          "uncompress␣-c"
          # elif (defined HAVE_COMPRESS)
          "compress␣-cd"
          # endif
          ;
#endif
#endif
```

This code is used in section 118.

122. Before testing whether the input file is compressed, see if the name we were given is a symbolic link. If so, follow the link and test the actual file. We only follow links up to 50 levels. We copy the file name given us to *jname*, then attempt to interpret it as a symbolic link by calling *readlink*, which will fail if the name is not, in fact, a symbolic link. If it is, we obtain the link destination as a C string, which is copied into *jname* prior to the test for a compressed file extension.

```
⟨ Check for symbolic link to compressed file 122 ⟩ ≡
#ifdef HAVE_READLINK
  int maxSlinks = 50;
  string jname = fname;
  char slbuf[1024];

  while (maxSlinks−− > 0) {
    int sll = readlink(jname.c_str( ), slbuf, (sizeof slbuf) − 1);
    if (sll ≥ 0) {
      assert(sll < static_cast⟨int⟩(sizeof slbuf));
      slbuf[sll] = 0;
      jname = slbuf;
    }
    else {
      break;
    }
  }
  if (maxSlinks ≤ 0) {
    cerr ≪ "Warning:␣probable␣symbolic␣link␣loop␣for␣\"" ≪ fname ≪ "\"" ≪ endl;
  }
#endif
```

This code is used in sections 118 and 126.

123. If our input file bears an extension which identifies it as a compressed file, we use *popen* to create a file handle connected to a pipe to the appropriate decompression program. The pipe is then screwed into the input stream from which we subsequently read.

```
⟨ Open pipe to read compressed file 123 ⟩ ≡
  string cmd(Uncompress_command);

  cmd += '␣' + fname;
  ip = popen(cmd.c_str( ), "r");
  isc.attach(fileno(ip));
  is = &isc;
```

This code is used in section 118.

124. Some mail systems define mail folders as directories containing individual messages as files. If the folder name is in fact a directory, set up to retrieve the contents of all the files it contains logically concatenated.

```
⟨ Check whether folder is a directory of messages 124 ⟩ ≡
#ifdef HAVE_DIRECTORY_TRAVERSAL
  dirFolder = false;
  struct stat fs;
  if ((stat(fname.c_str( ), &fs) ≡ 0) ∧ S_ISDIR(fs.st_mode)) {
    dh = opendir(fname.c_str( ));
    if (dh ≠ Λ) {
      dirFolder = true;
      dirName = fname;
      pathSeparator = '/';      /* Should detect in configuration process */
      if (¬findNextFileInDirectory(fname)) {
        nullstream.str("");
        is = &nullstream;      /* Doooh!!! No mail messages in directory */
      }
      else {
        if (verbose) {
          cerr ≪ "Processing␣files␣from␣directory␣\"" ≪ dirName ≪ "\"." ≪ endl;
        }
      }
    }
    else {
      cerr ≪ "Cannot␣open␣mail␣folder␣directory␣\"" ≪ fname ≪ "\"" ≪ endl;
      exit(1);
    }
  }
#endif
```

This code is cited in section 206.

This code is used in section 118.

125. When we're reading a mail folder consisting of a directory of individual mail messages, when we reach the end of a message file we wish to seamlessly advance to the next file, logically concatenating the files in the directory. This method, which should be called whenever the next file in the directory is required, searches the directory for the next eligible file and opens it. We return *true* if the file was opened successfully and *false* if the end of the directory was hit whilst looking for the next file.

```
⟨ Class implementations 9 ⟩ +≡
#ifdef HAVE_DIRECTORY_TRAVERSAL
  bool mailFolder::findNextFileInDirectory(string &fname)
  {
    assert(dirFolder);
    if (dh ≡ Λ) {
      return false;      /* End of directory already encountered */
    }
    while (true) {
      struct dirent *de;
      struct stat fs;
      de = readdir(dh);
      if (de ≡ Λ) {
        closedir(dh);
        dh = Λ;
        return false;
      }
      cfName = dirName + pathSeparator + de→d_name;
      if (stat(cfName.c_str( ), &fs) ≡ 0) {
        if (S_ISREG(fs.st_mode)) {
          fname = cfName;
          return openNextFileInDirectory( );
        }
      }
      else {
        if (verbose) {
          cerr ≪ "Cannot␣get␣status␣of␣" ≪ cfName ≪ ".␣␣Skipping." ≪ endl;
        }
      }
    }
  }
#endif
```

126. Open the next file in a directory of files which constitute a logical mail folder. *findNextFileInDirectory* has already vetted and expanded the path name, certifying that (at least when it checked) the target was an extant regular file.

```
⟨ Class implementations 9 ⟩ +≡
#ifdef HAVE_DIRECTORY_TRAVERSAL
  bool mailFolder :: openNextFileInDirectory(void)
  {
    assert(dirFolder);
    if (dh ≡ Λ) {
      return false;
    }
#ifdef COMPRESSED_FILES
    string fname = cfName;
    ⟨ Check for symbolic link to compressed file 122 ⟩;
    if (fname.rfind(Compressed_file_type) ≠ string :: npos) {
      string cmd(Uncompress_command);
      cmd += '␣' + fname;
      ip = popen(cmd.c_str(), "r");
      ifdir.attach(fileno(ip));
      ifdir.clear();      /* Stupid attach doesn't reset ios :: eofbit! */
    }
    else {
#endif
      ifdir.open(cfName.c_str());
      if (¬ifdir.is_open()) {
        if (verbose) {
          cerr ≪ "Unable␣to␣open␣mail␣folder␣directory␣file␣\"" ≪ cfName ≪ "\"" ≪ endl;
        }
        return false;
      }
#ifdef COMPRESSED_FILES
    }
#endif
    is = &ifdir;
    return true;
  }
#endif
```

127. When we hit end of file, check whether we're traversing a directory and, if so, advance to the next file within it. When we reach the end of the directory, call it quits.

```
⟨ Advance to next file if traversing directory 127 ⟩ ≡
#ifdef HAVE_DIRECTORY_TRAVERSAL
  if (dirFolder) {
    if (ip ≠ Λ) {
      pclose(ip);
      ip = Λ;
    }
    else {
      ifdir.close();     /* Close previous file from directory */
    }
    if (findNextFileInDirectory(cfName)) {
      continue;
    }
  }
#endif
```

This code is used in section 119.

128. Each message in a folder begins with a line containing the text "`From `" starting in the first column. (Lines within messages which match this pattern are quoted, usually by inserting a "`>`" character in column 1.) Here we check for the start of a new message. Upon finding one, we increment the number of messages in the folder, mark the start of a new message, and set the *inHeader* flag to indicate we're parsing the header section of the message.

```
#define messageSentinel "From␣"     /* First line of each message in folder */
⟨ Check for start of new message in folder 128 ⟩ ≡
  if (s.substr(0, (sizeof messageSentinel) − 1) ≡ messageSentinel) {
    nMessages++;
    newMessage = true;
    inHeader = true;
    multiPart = false;
    inPartHeader = false;
    partHeaderLines = 0;
    bodyContentType = bodyContentTypeCharset = bodyContentTransferEncoding = "";
    while (¬partBoundaryStack.empty()) {
      ostringstream os;
      os ≪ "Orphaned␣part␣boundary␣on␣stack:␣\"" ≪ partBoundaryStack.top() ≪ "\"";
      reportParserDiagnostic(os);
      partBoundaryStack.pop();
    }
    ⟨ Reset MIME decoder state 131 ⟩;
  }
  else {
    newMessage = false;
  }
```

This code is cited in section 206.

This code is used in section 119.

129. To facilitate message parsing, we delete any white space from the ends of lines. Mail transfer agents are explicitly permitted to do this, and all forms of encoding are proof against it.

```
⟨ Eliminate any trailing space from line 129 ⟩ ≡
  while ((s.length() > 0) ∧ (isISOspace(s[s.length() − 1]))) {
    s.erase(s.length() − 1);
  }
```

This code is used in section 119.

130. If we're within the message header section, there are various things we want to be on the lookout for. First, of course, is the blank line that denotes the end of the header. If the header declares the content type of the body to be MIME multi-part, we need to save the part boundary separator for later use. As it happens, this code works equally fine for parsing the part headers which follow the sentinel denoting the start of new part in a MIME multi-part message.

```
⟨ Process message header lines 130 ⟩ ≡
  if (inHeader ∨ inPartHeader) {
    if (s ≡ "") {
      if (inHeader) {
        if ((¬multiPart) ∧ (bodyContentTransferEncoding ≠ "")) {
          mimeContentType = bodyContentType;
          mimeContentTypeCharset = bodyContentTypeCharset;
          mimeContentTransferEncoding = bodyContentTransferEncoding;
          multiPart = true;
          partBoundary = "";
        }
      }
      inHeader = inPartHeader = false;
      ⟨ Activate MIME decoder if required 140 ⟩;
    }
    ⟨ Check for continuation of mail header lines 132 ⟩;
    ⟨ Process multipart MIME header declaration 137 ⟩;
    ⟨ Process body content type declarations 134 ⟩;
    ⟨ Check for encoded header line and decode 135 ⟩;
  }
```

This code is used in section 119.

131. At the end of a MIME part, switch off the decoder and reset the part properties to void.

```
⟨ Reset MIME decoder state 131 ⟩ ≡
  mimeContentType = mimeContentTypeCharset = mimeContentTypeBoundary =
      mimeContentTransferEncoding = "";
  mdp = Λ;
  mbi = Λ;
  asp = Λ;
  byteStream = false;
```

This code is used in sections 118, 119, and 128.

132. Statements in the message header section may be continued onto multiple lines. Continuations are denoted by white space in the first column of successive continuations. To simplify header parsing, we look ahead and concatenate all continuations into one single header statement. The twiddling with *lal* in the following code is to ensure the integrity of transcripts. We delete trailing space from the look ahead line before concatenating it, but if we in fact looked ahead to a line which is not a continuation, we want to eventually save it in the transcript as it originally arrived, complete with trailing space, so we replace it with the original line before deleting the trailing space.

```
⟨ Check for continuation of mail header lines 132 ⟩ ≡
  ⟨ Check for lines with our sentinel already present in the header 133 ⟩;
  while ((inHeader ∨ inPartHeader) ∧ getline(∗is, lookAheadLine) ≠ Λ) {
    string lal = lookAheadLine;
    while ((lookAheadLine.length( ) > 0) ∧ (isISOspace(lookAheadLine[lookAheadLine.length( ) − 1])))
      {
      lookAheadLine.erase(lookAheadLine.length( ) − 1);
    }
    if ((lookAheadLine.length( ) > 0) ∧ isISOspace(lookAheadLine[0])) {
      string :: size_typep = 1;
      while (isISOspace(lookAheadLine[p])) {
        p++;
      }
      s += lookAheadLine.substr(p);
      if ((tlist ≠ Λ) ∧ (¬isSpoofedHeader)) {
        tlist→push_back(lal);
      }
      continue;
    }
    lookedAhead = true;
    lookAheadLine = lal;
    break;
  }
  if (isSpoofedHeader) {
    ostringstream os;
    os ≪ "Spoofed␣header␣rejected:␣" ≪ s;
    reportParserDiagnostic(os.str( ));
    continue;
  }
```

This code is cited in section 206.

This code is used in section 130.

133. A clever junk mail author might try to evade filtering based on the header items we include in the `--transcript` by including his own, on the assumption that a downstream filter would not detect the multiple items and filter on the first one it found. To prevent this, and to make it more convenient when feeding transcripts back through the program (for testing the effects of different settings or for training on new messages), we detect header lines which begin with our *Xfile* sentinel and completely delete them from the transcript. The *isSpoofedHeader* flag causes continuation lines, if any, to be deleted as well. (At this writing we never use continuations of our header items, but better safe than sorry.)

⟨ Check for lines with our sentinel already present in the header 133 ⟩ ≡

```
bool isSpoofedHeader = false;
if (inHeader) {
  string sc = s, scx = Xfile;
  stringCanonicalise(sc);
  stringCanonicalise(scx);
  scx += '-';
  if (sc.substr(0, scx.length()) ≡ scx) {
    if (tlist ≠ Λ) {
      tlist→pop_back();
    }
    isSpoofedHeader = true;
  }
}
```

This code is cited in section 206.

This code is used in section 132.

134. It is possible for the main body of a message to be encoded with a `Content-Transfer-Encoding` specification. While encoding is usually encountered in MIME multi-part messages, junk mail sometimes takes advantage of encoding to hide trigger words from content-based filters. If the message body is encoded, we need to interpose the appropriate filter before parsing it.

⟨ Process body content type declarations 134 ⟩ ≡

```
{
  string arg, par;
  if (compareHeaderField(s, "content-type", arg)) {
    if (parseHeaderArgument(s, "charset", par)) {
      stringCanonicalise(par);
      bodyContentTypeCharset = par;
    }
    bodyContentType = arg;
  }
  if (inHeader ∧ compareHeaderField(s, "content-transfer-encoding", arg)) {
    bodyContentTransferEncoding = arg;
  }
}
```

This code is used in section 130.

135. Message header lines may contain sequences of characters encoded in `Quoted-Printable` or `Base64` form (since mail headers must not contain 8 bit characters). To better extract words from these lines, we test for such subsequences and replace them with the encoded text. Due to the fact that, in the fullness of time, this code will be fed every conceivable kind of nonconforming trash, it must be completely bulletproof. The flailing around with *p4* protects against falling into a loop when decoding a sequence fails.

```
⟨Check for encoded header line and decode 135⟩ ≡
  if (inHeader) {
    string sc = s;
    string::size_typep, p1, p2, p3, p4;
    char etype;
    unsigned int ndecodes = 0;
    string charset;
    stringCanonicalise(sc);
    p4 = 0;
    while (((p = sc.find("=?", p4)) ≠ string::npos)) {
      p4 = p + 2;
      if (((p1 = sc.find("?q?", p4)) ≠ string::npos) ∨ ((p1 = sc.find("?b?", p4)) ≠ string::npos))
        {
        charset = sc.substr(p4, p1 − p4);
        etype = sc[p1 + 1];
        p4 = p1 + 3;
        if ((p2 = sc.find("?=", p4)) ≠ string::npos) {
          p1 += 3;
          p3 = p2 − p1;
          string drt;
          if (etype ≡ 'q') {
            drt = quotedPrintableMIMEdecoder::decodeEscapedText(sc.substr(p1, p3), this);
          }
          else {
            assert(etype ≡ 'b');
            drt = base64MIMEdecoder::decodeEscapedText(sc.substr(p1, p3), this);
          }
          ⟨Interpret header quoted string if character set known 136⟩;
          sc.replace(p, (p2 − p) + 2, drt);
          p4 = p + drt.length();
          ndecodes++;
        }
      }
    }
    if (ndecodes > 0) {
      s = sc;
    }
  }
```

This code is cited in section 206.

This code is used in section 130.

136. After decoding the `Quoted-Printable` or `Base64` sequence from the header line, examine its character set specification. If it is a character set we know how to decode and interpret, instantiate the appropriate components and replace the decoded sequence with its interpretation. There is no need to further process `ISO-8859` sequences.

```
⟨Interpret header quoted string if character set known 136⟩ ≡
  if (charset.substr(0, 6) ≡ "gb2312") {
    EUC_MBCSdecoder mbd_euc;    /* EUC decoder */
    GB2312_MBCSinterpreter mbi_gb2312;    /* GB2312 interpreter */

    mbi_gb2312.setDecoder(mbd_euc);
    drt = mbi_gb2312.decodeLine(drt);
  }
  else if (charset ≡ "big5") {
    Big5_MBCSdecoder mbd_big5;    /* Big5 decoder */
    Big5_MBCSinterpreter mbi_big5;    /* Big5 interpreter */

    mbi_big5.setDecoder(mbd_big5);
    drt = mbi_big5.decodeLine(drt);
  }
  else if (charset ≡ "utf-8") {
    UTF_8_Unicode_MBCSdecoder mbd_utf_8;    /* Unicode UTF-8 decoder */
    Unicode_MBCSinterpreter mbi_unicode;    /* Unicode interpreter */

    mbi_unicode.setDecoder(mbd_utf_8);
    drt = mbi_unicode.decodeLine(drt);
  }
  else if (charset ≡ "euc-kr") {
    EUC_MBCSdecoder mbd_euc;    /* EUC decoder */
    KR_MBCSinterpreter mbi_kr;    /* Korean (euc-kr) interpreter */

    mbi_kr.setDecoder(mbd_euc);
    drt = mbi_kr.decodeLine(drt);
  }
  else if (charset.substr(0, 8) ≡ "iso-8859") {
    /* No decoding or interpretation required for ISO-8859 */
  }
  else {
    ostringstream os;

    os ≪ "Subject␣line:␣no␣interpreter␣for␣(" ≪ charset ≪ ")␣character␣set.";
    reportParserDiagnostic(os.str());
  }
```

This code is used in section 135.

137. A multi-part message in MIME format will contain a declaration in the header which identifies the body as being in that format and provides a part separator sentinel which appears before each subsequent part. We test for the MIME declaration and save the part boundary sentinel for later use.

⟨ Process multipart MIME header declaration 137 ⟩ ≡

```
string :: size_type p, p1 ;
string arg;
if (inHeader ∧ compareHeaderField(s, "content-type", arg)) {
  string sc = s;
  stringCanonicalise(sc);
  if ((p = sc.find("multipart/", 13)) ≠ string :: npos) {
    if ((p = sc.find("boundary=", p + 10)) ≠ string :: npos) {
      if (s[p + 9] ≡ '\"') {
        p1 = sc.find("\"", p + 10);
        p += 10;
      }
      else {
        p += 9;
        p1 = sc.length() − p;
      }
      multiPart = true;
      partBoundary = s.substr(p, (p1 − p));
      if (Annotate('d')) {
        ostringstream os;
        os ≪ "Multi-part␣boundary:␣\"" ≪ partBoundary ≪ "\"";
        reportParserDiagnostic(os);
      }
    }
  }
}
```

This code is cited in section 206.

This code is used in section 130.

138. If we're in the body of a MIME multi-part message, we must test each line against the *partBoundary* sentinel declared in the "`Content-type:`" header statement. If the line is a part boundary, we then must parse the part header which follows.

⟨ Check for MIME part sentinel 138 ⟩ ≡

```
if (multiPart ∧ (¬inHeader) ∧ (partBoundary ≠ "") ∧ (s.substr(0, 2) ≡ "--") ∧ (s.substr(2,
      partBoundary.length()) ≡ partBoundary) ∧ (s.substr(partBoundary.length() + 2) ≠ "--")) {
  inPartHeader = true;
  mimeContentType = mimeContentTypeCharset = mimeContentTypeBoundary =
      mimeContentTransferEncoding = "";
}
```

This code is used in section 119.

139. If we're in the body of text encoded in a multiple-byte character set, pass the text through the interpreter to convert it into a form we can better recognise.

```
⟨ Decode multiple byte character set 139 ⟩ ≡
  if ((mbi ≠ Λ) ∧ (¬inHeader) ∧ (¬inPartHeader)) {
    s = mbi→decodeLine(s);
  }
```

This code is used in section 119.

140. If we've just reached the end of a MIME part header, determine if the body which follows requires decoding. If so, activate the appropriate decoder and place it in the pipeline between the raw mail folder and our parsing code.

```
⟨ Activate MIME decoder if required 140 ⟩ ≡
  if (multiPart) {
    assert(mdp ≡ Λ);
#ifdef TYPE_LOG    /* If TYPE_LOG is defined, we create a file containing all of the part properties
            we've seen. You can obtain a list of things you may need to worry about by processing one of
            the fields n of this file with a command like cut -fn /tmp/typelog.txt | sort | uniq. */
    typeLog ≪ mimeContentType ≪ "\t" ≪ mimeContentTypeCharset ≪ "\t" ≪
            mimeContentTransferEncoding ≪ endl;
#endif
    ⟨ Check for change of sentinel within message 141 ⟩;
    ⟨ Check for application file types for which we have a decoder 142 ⟩;
    ⟨ Detect binary parts worth parsing for embedded ASCII strings 143 ⟩;
    ⟨ Test for Content-Types we always ignore 144 ⟩
    ⟨ Process Content-Types we are interested in parsing 145 ⟩;
  }
```

This code is cited in section 206.

This code is used in section 130.

141. The sentinel which delimits parts of a multi-part message may be changed in the middle of the message by a `Content-Type` of `multipart/alternative` specifying a new `boundary=`. Detect this and change the part boundary on the fly. These parts usually seem devoid of content, but just in case fake a content type of `text/plain` so anything which may be there gets looked at.

```
⟨ Check for change of sentinel within message 141 ⟩ ≡
  if (mimeContentType ≡ "multipart/alternative") {
    if (mimeContentTypeBoundary ≠ "") {
      partBoundaryStack.push(partBoundary);
      partBoundary = mimeContentTypeBoundary;
    }
    else {
      if (Annotate('d')) {
        ostringstream os;
        os ≪ "Boundary␣missing␣from␣Content-Type␣of␣multipart/alternative.";
        reportParserDiagnostic(os);
      }
    }
  }
```

This code is used in section 140.

142. We have decoders for certain application file types. Check the `Content-Type` for types we can decode, and if it's indeed one we can, splice the appropriate decoder into the pipeline.

```
⟨Check for application file types for which we have a decoder 142⟩ ≡
#ifdef HAVE_PDF_DECODER
  if (mimeContentType ≡ "application/pdf") {
    asp = &aspPdf;
  }
  else
#endif
    if ((mimeContentType ≡ "application/x-shockwave-flash") ∨ (mimeContentType ≡
          "image/vnd.rn-realflash")) {
      asp = &aspFlash;
    }
  if (asp ≠ Λ) {
    asp→setMailFolder(this);
    if (Annotate('d')) {
      ostringstream os;

      os ≪ "Activating␣" ≪ asp→name() ≪ "␣application␣file␣decoder.";
      reportParserDiagnostic(os);
    }
  }
```

This code is used in section 140.

143. Certain MIME `Content-Type` declarations denote binary files best classified by parsing them for ASCII strings. Test for such files and invoke the requisite decoder unless binary stream parsing has been disabled by setting *streamMinTokenLength* to zero or the file is already scheduled for parsing by an application-specific string parser.

```
⟨Detect binary parts worth parsing for embedded ASCII strings 143⟩ ≡
  if ((asp ≡ Λ) ∧ (streamMinTokenLength > 0) ∧ (mimeContentType.substr(0, 12) ≡ "application/"))
    {
    if (Annotate('d')) {
      ostringstream os;

      os ≪ "Activating␣byte␣stream␣parser␣for␣\"" ≪ mimeContentType ≪ "\"";
      reportParserDiagnostic(os);
    }
    byteStream = true;
  }
```

This code is used in section 140.

144. Test for Content-Types we are never interested in parsing, regardless of their encoding. This includes images, video, and most application specific files which UNIX `strings` would make no sense of. These parts are dispatched to the sink decoder for disposal. Note that some of these items may be compressed files and/or archives (`zip`, `gzip`, `tar`, etc.) which might be comprehensible if we could enlist the appropriate utilities, but we'll defer that refinement for now.

```
⟨ Test for Content-Types we always ignore 144 ⟩ ≡
  if (Annotate('d')) {
    ostringstream os;
    reportParserDiagnostic("");
    os ≪ "mimeContentType:␣{" ≪ mimeContentType ≪ "}";
    reportParserDiagnostic(os);
    os.str("");
    os ≪ "mimeContentTypeCharset:␣{" ≪ mimeContentTypeCharset ≪ "}";
    reportParserDiagnostic(os);
    os.str("");
    os ≪ "mimeContentTransferEncoding:␣{" ≪ mimeContentTransferEncoding ≪ "}";
    reportParserDiagnostic(os);
  }
  if ((asp ≡ Λ) ∧ (mimeContentType.substr(0, 6) ≡ "image/") ∨ (mimeContentType.substr(0,
        6) ≡ "video/")) {
    smd.set(is, this, partBoundary, tlist);
    mdp = &smd;
    if (Annotate('d')) {
      ostringstream os;
      os ≪ "Activating␣MIME␣sink␣decoder␣with␣sentinel:␣\"" ≪ partBoundary ≪
          "\"␣due␣to␣Content-Type␣=␣" ≪ mimeContentType;
      reportParserDiagnostic(os);
    }
    if (dlist) {
      dlist→push_back(Xfile + "-Decoder:␣Sink");
    }
  }
```

This code is used in section 140.

145. Next, check for content types we're always interested parsing. This includes most forms labeled as text and embedded mail messages. If the content is of interest but is encoded, make sure we have the requisite decoder and, if so, plumb it into the pipeline.

```
⟨ Process Content-Types we are interested in parsing 145 ⟩ ≡
  else
    if (byteStream ∨ (asp ≠ Λ) ∨ (mimeContentType ≡ "plain/txt") ∨ (mimeContentType.substr(0,
          5) ≡ "text/") ∨ (mimeContentType ≡ "message/rfc822")) {
      ⟨ Test for multiple byte character sets and activate decoder if available 146 ⟩;
      ⟨ Verify Content-Transfer-Encoding and activate decoder if necessary 147 ⟩;
      ⟨ Cancel byte stream interpretation for non-binary encoded parts 148 ⟩;
    }
```

This code is used in section 140.

146. Just because we're *interested* in the contents of this part, doesn't necessarily mean we can *comprehend* it. First of all, it must be encoded in a form we can either read directly or have a decoder for, and secondly it must be in a character set we understand, not some Asian chicken tracks. First of all, test the character set and accept only those we read directly or have interpreters for.

```
⟨ Test for multiple byte character sets and activate decoder if available 146 ⟩ ≡
  bool gibberish = false;
  if (mimeContentTypeCharset.substr(0, 6) ≡ "gb2312") {
    mbi_gb2312.setDecoder(mbd_euc);
    mbi = &mbi_gb2312;
  }
  if (mimeContentTypeCharset ≡ "big5") {
    mbi_big5.setDecoder(mbd_big5);
    mbi = &mbi_big5;
  }
  if (mimeContentTypeCharset ≡ "utf-8") {
    mbi_unicode.setDecoder(mbd_utf_8);
    mbi = &mbi_unicode;
  }
  if (mimeContentTypeCharset ≡ "euc-kr") {
    mbi_kr.setDecoder(mbd_euc);
    mbi = &mbi_kr;
  }
#ifdef CHECK_FOR_GIBBERISH_CHARACTER_SETS
  if ((mimeContentTypeCharset.length() ≡ 0) ∨ (mimeContentTypeCharset ≡
        "us-ascii") ∨ (mimeContentTypeCharset.substr(0,
        8) ≡ "iso-8859") ∨ (mimeContentTypeCharset ≡ "windows-1251")) {
    if (Annotate('d')) {
      ostringstream os;
      os ≪ "Accepting␣part␣in␣Content-Type-Charset:␣" ≪ mimeContentTypeCharset ≪
          "␣␣(" ≪ mimeContentType ≪ "␣" ≪ mimeContentTransferEncoding ≪ ")";
      reportParserDiagnostic(os);
    }
  }
  else {
    if (Annotate('d')) {
      ostringstream os;
      os ≪ "Rejecting␣part␣in␣Content-Type-Charset:␣" ≪ mimeContentTypeCharset ≪
          "␣␣(" ≪ mimeContentType ≪ "␣" ≪ mimeContentTransferEncoding ≪ ")";
      reportParserDiagnostic(os);
    }
    gibberish = true;
  }
#endif
```

This code is used in section 145.

147. If the contents appear to be in a character set we understand, we still aren't home free—the part may be encoded in a manner for which we lack a decoder. Analyse the `Content-Transfer-Encoding` specification and select the appropriate decoder. If we lack a decoder, we must regretfully consign the part to the sink decoder.

If we end up accreting any additional decoders, this should probably be re-written to look up the decoder in a **map**⟨**string**, **MIMEdecoder** $*$⟩ and use common code for every decoder.

⟨ Verify Content-Transfer-Encoding and activate decoder if necessary 147 ⟩ ≡

```
if (¬gibberish) {
  if ((mimeContentTransferEncoding.length() ≡ 0) ∨ (mimeContentTransferEncoding.substr(0,
        4) ≡ "7bit") ∨ (mimeContentTransferEncoding.substr(0,
        4) ≡ "8bit") ∨ (mimeContentTransferEncoding ≡ "ascii")) {
    imd.set(is, this, partBoundary, tlist);     /* Identity */
    mdp = &imd;
  }
  else if (mimeContentTransferEncoding ≡ "base64") {
    bmd.set(is, this, partBoundary, tlist);     /* Base64 */
    mdp = &bmd;
  }
  else if (mimeContentTransferEncoding ≡ "quoted-printable") {
    qmd.set(is, this, partBoundary, tlist);     /* Quoted-Printable */
    mdp = &qmd;
  }
  else {
    gibberish = true;
    smd.set(is, this, partBoundary, tlist);     /* Sink */
    mdp = &smd;
  }
  assert(mdp ≠ Λ);
  if (Annotate('d')) {
    ostringstream os;

    os ≪ (gibberish ? "Rejecting" : "Accepting") ≪
        "␣part␣in␣Content-Transfer-Encoding:␣" ≪ mimeContentTransferEncoding ≪
        "␣␣(" ≪ mimeContentTypeCharset ≪ "␣" ≪ mimeContentType ≪ ")";
    reportParserDiagnostic(os);
  }
  if (dlist) {
    dlist→push_back(Xfile + "-Decoder:␣" + mdp→name());
  }
  if (Annotate('d')) {
    ostringstream os;

    os ≪ "Activating␣MIME␣" ≪ mdp→name() ≪ "␣decoder␣with␣sentinel:␣" ≪ partBoundary;
    reportParserDiagnostic(os);
  }
}
```

This code is cited in section 206.

This code is used in section 145.

148. If we think we're about to process a byte stream, but it isn't actually encoded, think again and treat the content as regular text, which it in all likelihood actually is.

```
⟨ Cancel byte stream interpretation for non-binary encoded parts 148 ⟩ ≡
  if (byteStream ∧ (mdp ≡ Λ)) {
    if (Annotate('d')) {
      ostringstream os;
      os ≪ "Canceling␣byte␣stream␣mode␣due␣to␣Content-Transfer-Encoding:␣{" ≪
          mimeContentTransferEncoding ≪ "}␣␣(" ≪ mimeContentTypeCharset ≪ "␣" ≪
          mimeContentType ≪ ")";
      reportParserDiagnostic(os);
    }
    byteStream = false;
  }
```

This code is used in section 145.

149. Canonicalise a string in place to all lower-case characters.

```
⟨ Class implementations 9 ⟩ +≡
  void mailFolder :: stringCanonicalise(string &s)
  {
    for (unsigned int i = 0; i < s.length( ); i++) {
      if (isISOupper(s[i])) {
        s[i] = toISOlower(s[i]);
      }
    }
  }
```

150. To facilitate parsing of header fields, this static method performs a case-insensitive test for header field *target* and, if it is found, stores its argument into *arg*, set to canonical lower case.

```
⟨ Class implementations 9 ⟩ +≡
  bool mailFolder :: compareHeaderField(string &s, const string target, string &arg)
  {
    if (s.length() > target.length()) {
      string sc = s;
      stringCanonicalise(sc);
      if ((sc.substr(0, target.length()) ≡ target) ∧ (sc[target.length()] ≡ ':')) {
        unsigned int i;
        for (i = target.length() + 1; i < sc.length(); i++) {
          if (¬isISOspace(sc[i])) {
            break;
          }
        }
        if (i < sc.length()) {
          int n = 0;
          while ((i + n) < sc.length()) {
            if (isISOspace(sc[i + n]) ∨ (sc[i + n] ≡ ';')) {
              break;
            }
            n++;
          }
          arg = sc.substr(i, n);
        }
        else {
          arg = "";
        }
        return true;
      }
    }
    return false;
  }
```

151. This static method tests for an argument to a header field and stores the argument, if present, into *arg*. The argument name is canonicalised to lower case, but the argument is left as-is. Quotes are deleted from quoted arguments.

```
⟨Class implementations 9⟩ +≡
  bool mailFolder::parseHeaderArgument(string &s, const string target, string &arg)
  {
    if (s.length() > target.length()) {
      string sc = s;
      string::size_type p, p1;
      stringCanonicalise(sc);
      if (((p = sc.find(target)) ≠ string::npos) ∧ (sc.length() >
            (p + target.length())) ∧ (sc[p + target.length()] ≡ '=')) {
        p += target.length() + 1;
        if (p < s.length()) {
          if (s[p] ≡ '"') {
            if ((p1 = s.find('"', p + 1)) ≠ string::npos) {
              arg = s.substr(p + 1, p1 − (p + 1));
              return true;
            }
          }
          else {
            string::size_type i = p;
            for ( ; i < s.length(); i++) {
              if (¬isISOspace(s[i])) {
                break;
              }
            }
            if (i < s.length()) {
              int n = 0;
              while ((i + n) < s.length()) {
                if ((isISOspace(s[i + n])) ∨ (s[i + n] ≡ ';')) {
                  break;
                }
                n++;
              }
              arg = s.substr(i, n);
            }
            else {
              arg = "";
            }
            return true;
          }
        }
      }
    }
    return false;
  }
```

152. Here we parse interesting fields from a MIME message part header.

⟨ Parse MIME part header 152 ⟩ ≡

```
if (multiPart ∧ inPartHeader) {
   string arg, par;
   partHeaderLines++;
   if (compareHeaderField(s, "content-type", arg)) {
      if (parseHeaderArgument(s, "charset", par)) {
         stringCanonicalise(par);
         mimeContentTypeCharset = par;
      }
      if (parseHeaderArgument(s, "boundary", par)) {
         mimeContentTypeBoundary = par;
      }
      mimeContentType = arg;
   }
   if (compareHeaderField(s, "content-transfer-encoding", arg)) {
      mimeContentTransferEncoding = arg;
   }
}
```

This code is used in section 119.

153. Write the message transcript saved in *tlist* to the designated file name *fname*. If *fname* is “-”, the transcript is written to standard output.

⟨ Class implementations 9 ⟩ +≡

```
void mailFolder :: writeMessageTranscript(ostream &os = cout)
{
   assert(tlist ≠ Λ);
   unsigned int n = tlist→size( );
   if ((n > 1) ∧ (tlist→back( ).substr(0, (sizeof messageSentinel) − 1) ≡ messageSentinel)) {
      n−−;
   }
   list⟨string⟩ :: iteratorp = tlist→begin( );
   for (unsigned int i = 0; i < n; i++) {
      os ≪ *p++ ≪ endl;
   }
}
void mailFolder :: writeMessageTranscript(const string fname = "-")
{
   if (fname ≠ "-") {
      ofstream of(fname.c_str( ));
      writeMessageTranscript(of);
      of.close( );
   }
   else {
      writeMessageTranscript(cout);
   }
}
```

154. When we detect an error within the message, it's reported to standard error if we're in *verbose* mode and appended to the *parserDiagnostics* for inclusion in the transcript if the "p" annotation is selected. This method is **public** so higher-level parsing routines can use it to append their own diagnostics. Since in many cases we compose the diagnostic in an **ostringstream**, we overload a variant which accepts one directly as an argument.

```
⟨ Class implementations 9 ⟩ +≡
  void mailFolder::reportParserDiagnostic(const string s) const
  {
    if (verbose) {
      cerr ≪ s ≪ endl;
    }
    if (Annotate('p') ∨ Annotate('d')) {
      parserDiagnostics.push(s);
    }
  }
  void mailFolder::reportParserDiagnostic(const ostringstream &os) const
  {
    reportParserDiagnostic(os.str());
  }
```

155. Token definition.

A *tokenDefinition* object provides the means by which the *tokenParser* (below) distinguishes tokens in a stream of text. Tokens are defined by three arrays, each indexed by ISO character codes between 0 and 255. The first, *isToken*, is *true* for characters which comprise tokens. The second, *notExclusively*, is *true* for characters which may appear in tokens, but only in the company of other characters. The third, *notAtEnd* is *true* for characters which may appear within a token, but not at the start or the end of one.

```
⟨ Class definitions 8 ⟩ +≡
  class tokenDefinition {
protected:
  static const int numTokenChars = 256;
  bool isToken[numTokenChars], notExclusively[numTokenChars], notAtEnd[numTokenChars];
  unsigned int minTokenLength, maxTokenLength;
public:
  tokenDefinition( )
  {
    clear( );
  }
  void clear(void)
  {
    for (int i = 0; i < numTokenChars; i++) {
      isToken[i] = notExclusively[i] = notAtEnd[i] = false;
    }
    setLengthLimits(1, 65535);
  }
  void setLengthLimits(unsigned int lmin = 0, unsigned int lmax = 0)
  {
    if (lmin > 0) {
      minTokenLength = lmin;
    }
    if (lmax > 0) {
      maxTokenLength = lmax;
    }
  }
  unsigned int getLengthMin(void) const
  {
    return minTokenLength;
  }
  unsigned int getLengthMax(void) const
  {
    return maxTokenLength;
  }
  bool isTokenMember(const int c) const
  {
    assert(c ≥ 0 ∧ c < numTokenChars);
    return isToken[c];
  }
  bool isTokenNotExclusively(const int c) const
  {
    assert(c ≥ 0 ∧ c < numTokenChars);
    return $notExclusively[c]$;
  }
  bool $isTokenNotAtEnd$(const int $c$) const
  {
    assert($c \geq 0 \land c < numTokenChars$);
    return $notAtEnd[c]$;
  }
  bool $isTokenLengthAcceptable$(string :: $size\_type\ l$)const
      {
        return $(l \geq minTokenLength) \land (l \leq maxTokenLength)$;
      }
      bool $isTokenLengthAcceptable$(const string $t$) const
      {
        return $isTokenLengthAcceptable(t.length())$;
      }
      void $setTokenMember$(bool $v$, const int $cstart$, const int $cend = -1$)
      {
        assert($cstart \geq 0 \land cstart \leq numTokenChars$);
        assert($(cend \equiv -1) \lor (cend \geq cstart \land cend \leq numTokenChars)$);
        for (int $i = cstart$; $i \leq cend$; $i$++) {
          $isToken[i] = v$;
        }
      }
      void $setTokenNotExclusively$(bool $v$, const int $cstart$, const int $cend = -1$)
      {
        assert($cstart \geq 0 \land cstart \leq numTokenChars$);
        assert($(cend \equiv -1) \lor (cend \geq cstart \land cend \leq numTokenChars)$);
        for (int $i = cstart$; $i \leq cend$; $i$++) {
          $notExclusively[i] = v$;
        }
      }
      void $setTokenNotAtEnd$(bool $v$, const int $cstart$, const int $cend = -1$)
      {
        assert($cstart \geq 0 \land cstart \leq numTokenChars$);
        assert($(cend \equiv -1) \lor (cend \geq cstart \land cend \leq numTokenChars)$);
        for (int $i = cstart$; $i \leq cend$; $i$++) {
          $notAtEnd[i] = v$;
        }
      }
      void $setISO\_8859defaults$(unsigned int $lmin = 0$, unsigned int $lmax = 0$);
      void $setUS\_ASCIIdefaults$(unsigned int $lmin = 0$, unsigned int $lmax = 0$); } ;
```

156. Initialise a **tokenDefinition** for parsing ISO-8859 text with our chosen defaults for punctuation embedded in such tokens. Any pre-existing definitions are cleared.

```
⟨ Class implementations 9 ⟩ +≡
  void tokenDefinition :: setISO_8859defaults(unsigned int lmin = 0, unsigned int lmax = 0)
  {
    clear( );
    setLengthLimits(lmin, lmax);
    for (unsigned int c = 0; c < 256; c++) {
      isToken[c] = (isascii(c) ∧ isdigit(c)) ∨ isISOalpha(c) ∨ (c ≡ '-') ∨ (c ≡ '\'') ∨ (c ≡ '$');
      notExclusively[c] = (isdigit(c) ∨ (c ≡ '-')) ? 1 : 0;
    }
    notAtEnd['-'] = notAtEnd['\''] = true;
  }
```

157. Initialise a **tokenDefinition** for parsing US-ASCII text with our chosen defaults for punctuation embedded in such tokens. Any pre-existing definitions are cleared.

```
⟨ Class implementations 9 ⟩ +≡
  void tokenDefinition :: setUS_ASCIIdefaults(unsigned int lmin = 0, unsigned int lmax = 0)
  {
    clear( );
    setLengthLimits(lmin, lmax);
    for (unsigned int c = 0; c < 128; c++) {
      isToken[c] = isalpha(c) ∨ isdigit(c);
      notExclusively[c] = (isdigit(c) ∨ (c ≡ '-')) ? 1 : 0;
    }
    isToken['_'] = notExclusively['_'] = true;
    notAtEnd['-'] = notAtEnd['\''] = true;
  }
```

158. Token parser.

A *tokenParser* reads lines from a **mailFolder** and returns tokens as defined by its active **tokenDefinition**. Separate **tokenDefinition**s can be defined for use while parsing regular text and binary byte streams, respectively. A *tokenParser* has the ability to save the lines parsed from a message in a *messageQueue*, permitting further subsequent analysis. Note that what is saved is "what the parser saw"—after MIME decoding or elision of ignored parts.

```
⟨ Class definitions 8 ⟩ +≡
  class tokenParser {
  protected:
    mailFolder *source;
    string cl;
    string::size_type clp;
    bool atEnd, inHTML, inHTMLcomment;
    tokenDefinition *td;      /* Token definition for text mode */
    tokenDefinition *btd;     /* Token definition for byte stream parsing */
    bool saveMessage;     /* Save current message in messageQueue ? */
    bool assemblePhrases;     /* Are we assembling phrases ? */
    deque⟨string⟩ phraseQueue;     /* Phrase assembly queue */
    deque⟨string⟩ pendingPhrases;     /* Queue of phrases awaiting return */
  public:
    list⟨string⟩ messageQueue;     /* Current message */
    tokenParser( )
    {
      td = Λ;
    }
    void setSource(mailFolder &mf)
    {
      source = &mf;
      cl = "";
      clp = 0;
      atEnd = inHTML = inHTMLcomment = false;
      saveMessage = false;
      messageQueue.clear( );
      phraseQueue.clear( );
      pendingPhrases.clear( );
      ⟨ Check phrase assembly parameters and activate if required 164 ⟩;
    }
    void setTokenDefinition(tokenDefinition &t, tokenDefinition &bt)
    {
      td = &t;
      btd = &bt;
    }
    void setTokenLengthLimits(unsigned int lMax, unsigned int lMin = 1, unsigned int
            blMax = 1, unsigned int blMin = 1)
    {
      assert(td ≠ Λ);
      td→setLengthLimits(lMin, lMax);
      assert(btd ≠ Λ);
      btd→setLengthLimits(blMin, blMax);
    }
    unsigned int getTokenLengthMin(void) const
    {
      return td→getLengthMin( );
    }
    unsigned int getTokenLengthMax(void) const
    {
      return td→getLengthMax( );
    }
    void reportParserDiagnostic(const string s) const
    {
      assert(source ≠ Λ);
      source→reportParserDiagnostic(s);
    }
    void reset(void)
    {
      if (inHTML) {
        reportParserDiagnostic("<HTML>␣tag␣unterminated␣at␣end␣of␣message.");
      }
      if (inHTMLcomment) {
        reportParserDiagnostic("HTML␣comment␣unterminated␣at␣end␣of␣message.");
      }
      inHTML = inHTMLcomment = false;
      clearMessageQueue( );
      phraseQueue.clear( );
      pendingPhrases.clear( );
    }
    bool nextToken(dictionaryWord &d);
    void assembleAllPhrases(dictionaryWord &d);
    ⟨ Message queue utilities 167 ⟩;
    bool isNewMessage(void) const
    {
      return atEnd ∨ (source→isNewMessage( ));
    }
  private:
    void nextLine(void)
    {
      while (true) {
        if (¬(source→nextLine(cl))) {
          atEnd = true;
          cl = "";
          break;
        }
        if (saveMessage) {
          messageQueue.push_back(cl);
        }
        if (source→isNewMessage( )) {
          reset( );
        }
        break;
      }
```

```
        clp = 0;
    }
};
```

159. The *nextToken* method stores the next token from the input source into its dictionary word argument and returns *true* if a token was found or *false* if the end of the input source was encountered whilst scanning for the next token.

```
#define ChIx(c) (static_cast⟨unsigned int⟩((c)) & #FF)
⟨ Class implementations 9 ⟩ +≡
  bool tokenParser::nextToken(dictionaryWord &d)
  {
    string token;
    while (¬atEnd) {
      ⟨ Check for assembled phrases in queue and return next if so 160 ⟩;
      token = "";
      string::size_type necount = 0;
      if (source→isByteStream()) {
        ⟨ Parse plausible tokens from byte stream 163 ⟩;
      }   /* Ignore non-token characters until start of next token */
      while ((clp < cl.length()) ∧ (inHTMLcomment ∨ (¬(td→isTokenMember(ChIx(cl[clp])))))) {
        ⟨ Check for HTML comments and ignore them 161 ⟩;
        ⟨ Check for within HTML content 162 ⟩;
        clp++;
      }   /* If end of line encountered before token start, advance to next line */
      if (clp ≥ cl.length()) {
        nextLine();
        continue;
      }   /* Check for characters we don't accept as the start of a token */
      if (td→isTokenNotAtEnd(ChIx(cl[clp]))) {
        clp++;
        continue;
      }   /* First character of token recognised; store and scan balance */
      if (td→isTokenNotExclusively(ChIx(cl[clp]))) {
        necount++;
      }
      token += cl[clp++];
      while ((clp < cl.length())) {
        if ((¬inHTMLcomment) ∧ (td→isTokenMember(ChIx(cl[clp])))) {
          if (td→isTokenNotExclusively(ChIx(cl[clp]))) {
            necount++;
          }
          token += cl[clp++];
        }
        else {
          ⟨ Check for HTML comments and ignore them 161 ⟩;
          if (inHTMLcomment) {
            clp++;
            continue;
          }
          break;
        }
      }   /* Prune characters we don't accept at the end of a token */
      while ((token.length() > 0) ∧ td→isTokenNotAtEnd(ChIx(token[token.length() − 1]))) {
        token.erase(token.length() − 1);
      }   /* Verify that the token meets our minimum and maximum length constraints */
```

```
          if (¬(td→isTokenLengthAcceptable(token))) {
            continue;
          }      /* We've either hit the end of the line or encountered a character that's not considered
                   part of a token. Return the token, leaving the class variables ready to carry on finding
                   the next token when we're called again. But first, if the token is composed entirely of
                   characters in the not_entirely class, we discard it. */
          if (necount ≡ token.length()) {
            continue;
          }
          d.set(token);
          d.toLower();      /* Convert to canonical form */
          ⟨Check for phrase assembly and generate phrases as required 165⟩;
          if (pTokenTrace ∧ saveMessage) {
            messageQueue.push_back(string("␣␣\"") + d.text + "\"");
          }
          return true;
        }
        return false;
      }
```

160. If we're assembling phrases, there may be one or more already assembled phrases sitting in the *pendingPhrases* queue. If so, remove it from the queue and return it.

```
⟨Check for assembled phrases in queue and return next if so 160⟩ ≡
  if (¬pendingPhrases.empty()) {
    token = pendingPhrases.front();
    pendingPhrases.pop_front();
    d.set(token);
    d.toLower();
    if (pTokenTrace ∧ saveMessage) {
      messageQueue.push_back(string("␣␣\"") + d.text + "\"");
    }
    return true;
  }
```

This code is used in section 159.

161. We wish to skip comments in HTML inclusions in mail, as junk mail frequently uses void HTML comments to break up trigger words for detectors. Strictly speaking, a space (or end of line) is required after the HTML begin comment and before the end comment delimiters, but most browsers don't enforce this and real-world HTML frequently violates this rule. So, we treat any sequence of characters between the delimiters as an HTML comment.

```
#define HTMLCommentBegin "<!--"   /* HTML comment start sentinel */
#define HTMLCommentEnd "-->"   /* HTML comment end sentinel */
```

⟨ Check for HTML comments and ignore them 161 ⟩ ≡

```
  if (inHTML ∧ ¬inHTMLcomment ∧ (cl.substr(clp, 4) ≡ HTMLCommentBegin)) {
    inHTMLcomment = true;
    clp += 4;   /* Skip over first HTML comment sentinel */
#ifdef HTML_COMMENT_DEBUG
    cout ≪ "------------------------------␣HTML␣Comment␣begin:␣" ≪ cl ≪ endl;
#endif
    continue;
  }
  if (inHTML ∧ inHTMLcomment ∧ (cl.substr(clp, 3) ≡ HTMLCommentEnd)) {
    inHTMLcomment = false;
    clp += 3;
#ifdef HTML_COMMENT_DEBUG
    cout ≪ "------------------------------␣HTML␣Comment␣end:␣" ≪ cl ≪ endl;
#endif
    continue;
  }
#ifdef HTML_COMMENT_DEBUG
  if (inHTMLcomment) {
    cout ≪ cl[clp];
    if (clp ≡ (cl.length() − 1)) {
      cout ≪ endl;
    }
  }
#endif
```

This code is used in section 159.

162. To avoid accidentally blundering into HTML comment discarding in non-HTML text, we look for start and end HTML tags and only activate HTML comment detection inside something which is plausibly HTML. Note that unclosed HTML tags and comments are automatically closed out when *reset* is called at the start of a new message from the mail folder.

```
⟨ Check for within HTML content 162 ⟩ ≡
  if (cl[clp] ≡ '<' ∧ (clp ≤ (cl.length( ) − 6))) {
    if ((cl[clp + 1] ≡ 'H' ∨ cl[clp + 1] ≡ 'h') ∧ (cl[clp + 5] ≡ '>' ∨ cl[clp + 5] ≡ '␣')) {
      string tag;
      for (int i = 1; i < 5; i++) {
        tag += (islower(cl[clp + i])) ? toupper(cl[clp + i]) : cl[clp + i];
      }
      if (tag ≡ "HTML") {
        inHTML = true;
#ifdef HTML_COMMENT_DEBUG
        cout ≪ "-------------------------------␣In␣HTML:␣" ≪ cl ≪ endl;
#endif
      }
    }
  }
  if (cl[clp] ≡ '<' ∧ (clp ≤ (cl.length( ) − 7))) {
    if ((cl[clp + 1] ≡ '/') ∧ (cl[clp + 2] ≡ 'H' ∨ cl[clp + 2] ≡ 'h') ∧ (cl[clp + 6] ≡ '>')) {
      string tag;
      for (int i = 2; i < 6; i++) {
        tag += (islower(cl[clp + i])) ? toupper(cl[clp + i]) : cl[clp + i];
      }
      if (tag ≡ "HTML") {
        inHTML = false;
#ifdef HTML_COMMENT_DEBUG
        cout ≪ "-------------------------------␣Out␣of␣HTML:␣" ≪ cl ≪ endl;
#endif
      }
    }
  }
```

This code is used in section 159.

163. If the item being read from the **mailFolder** has been identified as a binary byte stream, read it character by character and parse for probable strings. We use the byte stream **tokenDefinition** *btd* to determine token composition, permitting stricter construction of plausible tokens in binary byte streams.

We get here only when our *source* identifies itself as chewing through a byte stream with *isByteStream*. While in a byte stream, the **mailFolder** permits calls to its *nextByte* method, which returns bytes directly from the active stream decoder. At the end of the stream (usually denoted by the end sentinel of the MIME part containing the stream), *nextByte* returns −1 and clears the byte stream indicator. We escape from here when that happens, and go around the main loop in *nextToken* again, which will, now that byte stream mode is cleared, resume dealing with the mail folder at the *nextLine* level, where all of the housekeeping related to the end of the byte stream will be dealt with.

This code is so similar to the main loop it's embedded in it should probably be abstracted out as a token recogniser engine parameterised by the means of obtaining bytes and the token definition it applies. I may get around to this when I'm next in clean freak mode, but for the nonce I'll leave it as-is until I'm sure no additional special pleading is required when cracking byte streams.

```
⟨ Parse plausible tokens from byte stream 163 ⟩ ≡
  int b;

  while ((b = source→nextByte( )) ≥ 0) {
      /∗ Ignore non-token characters until start of next token ∗/
    if (¬(btd→isTokenMember(b))) {
      continue;
    }    /∗ Check for characters we don't accept as the start of a token ∗/
    if (btd→isTokenNotAtEnd(b)) {
      continue;
    }    /∗ First character of token recognised; store and scan balance ∗/
    if (btd→isTokenNotExclusively(b)) {
      necount++;
    }
    token += static_cast⟨char⟩(b);
    while (((b = source→nextByte( )) ≥ 0) ∧ btd→isTokenMember(b)) {
      if (btd→isTokenNotExclusively(b)) {
        necount++;
      }
      token += static_cast⟨char⟩(b);
    }    /∗ Prune characters we don't accept at the end of a token ∗/
    while ((token.length( ) > 0) ∧ btd→isTokenNotAtEnd(ChIx(token[token.length( ) − 1]))) {
      token.erase(token.length( ) − 1);
    }    /∗ Verify that the token meets our minimum and maximum length constraints ∗/
    if (¬(btd→isTokenLengthAcceptable(token))) {
      token = "";
      continue;
    }    /∗ Verify that the token isn't composed exclusively of characters permitted in a token but
           not allowed to comprise it in entirety. ∗/
    if (necount ≡ token.length( )) {
      token = "";
      continue;
    }
    d.set(token);
    d.toLower( );     /∗ Convert to canonical form ∗/
    ⟨ Check for phrase assembly and generate phrases as required 165 ⟩;
    if (pTokenTrace ∧ saveMessage) {
      messageQueue.push_back(string("␣␣\"") + d.text + "\"");
    }
    **return** *true*;
  }
  **continue**;
```

This code is used in section 159.

164. If the user has so requested, we can assemble tokens into phrases in a given length range. The default minimum and maximum length phrase is 1 word, which causes individual tokens to be returned as they are parsed. When the maximum is greater than one word, consecutive tokens (but never crossing a *reset* or *setSource* boundary) are assembled into phrases and output as pseudo-tokens of each length from the minimum to maximum length phrase.

Here we examine the phrase length parameters, report any erroneous specifications, and determine whether phrase assembly is required at all.

⟨ Check phrase assembly parameters and activate if required 164 ⟩ ≡

```
  assemblePhrases = false;
  if ((phraseMin ≠ 1) ∨ (phraseMax ≠ 1)) {
    if ((phraseMin ≥ 1) ∧ (phraseMax ≥ phraseMin)) {
      if ((phraseLimit > 0) ∧ (phraseLimit < ((phraseMax * 2) − 1))) {
        cerr ≪ "Invalid␣--phraselimit␣setting.␣␣Too␣small␣for␣specified␣--phrasemax." ≪
            endl;
      }
      else {
        assemblePhrases = true;
      }
    }
    else {
      cerr ≪ "Invalid␣--phrasemin/max␣parameters.␣␣Must␣be␣1␣<=␣min␣<=␣max." ≪ endl;
    }
  }
```

This code is used in section 158.

165. When *assemblePhrases* is set, each arriving token is used to generate all phrases including itself and previous tokens within the specified phrase length limits. Check for phrase assembly and invoke the *assembleAllPhrases* method if required.

⟨ Check for phrase assembly and generate phrases as required 165 ⟩ ≡

```
  if (assemblePhrases) {
    assembleAllPhrases(d);
    continue;
  }
```

This code is used in sections 159 and 163.

166. If we're assembling phrases, we take each token parsed (which has already been stored into the **dictionaryWord** argument *d* in canonical form) and place it on the *phraseQueue* queue, removing the element at the tail if the queue is longer than *phraseMax*. Then, if the queue contains *phraseMin* elements or more, iterate over the range of phrase lengths we wish to generate, creating phrases and storing them onto *pendingPhrases* for subsequent return.

```
⟨ Class implementations 9 ⟩ +≡
  void tokenParser :: assembleAllPhrases(dictionaryWord &d)
  {
    phraseQueue.push_back(d.text);
    if (phraseQueue.size( ) > phraseMax) {
      phraseQueue.pop_front( );
      assert(phraseQueue.size( ) ≡ phraseMax);
    }
    for (unsigned int p = phraseMin; p ≤ phraseMax; p++) {
      if (p ≤ phraseQueue.size( )) {
        deque⟨string⟩ :: const_reverse_iterator wp = phraseQueue.rbegin( );
        string phrase = "";
        for (unsigned int i = 0; i < p; i++) {
          phrase = (∗wp) + ((phrase ≡ "") ? "" : "␣") + phrase;
          wp++;
        }
        if ((phraseLimit ≡ 0) ∨ (phrase.length( ) ≤ phraseLimit)) {
          pendingPhrases.push_back(phrase);
        }
      }
    }
  }
```

167. The *messageQueue* can be used to store the lines of a message: "what the parser saw," after MIME decoding (but not elision of HTML comments or other processing in the parser itself). This is handy when debugging the lower level stuff. To enable saving messages in the queue, call *setSaveMessage* with an argument of *true*. The contents of *messageQueue* may be examined directly (it is a **public** member of the class), or written to an **ostream** with *writeMessageQueue*. One little detail—if you examine the *messageQueue* after the start of the next message in a folder has been detected, the first line of the next message will be the last item in the message queue; *writeMessageQueue* understands this and doesn't write the line, but if you're looking at the queue yourself it's up to you to cope with this.

```
⟨ Message queue utilities 167 ⟩ ≡
  void setSaveMessage(bool v)
  {
    saveMessage = v;
    source→setDiagnosticList(saveMessage ? (&messageQueue) : Λ);
  }
  bool getSaveMessage(void) const
  {
    return saveMessage;
  }
  void clearMessageQueue(void)
  {
    if (saveMessage) {
      string s;
      if (isNewMessage()) {
        s = messageQueue.back();
      }
      messageQueue.clear();
      if (isNewMessage()) {
        messageQueue.push_back(s);
      }
    }
  }
  void writeMessageQueue(ostream &os)
  {
    list⟨string⟩::size_typel = messageQueue.size(), n = 0;
    for (list⟨string⟩::iteratorp = messageQueue.begin(); p ≠ messageQueue.end(); p++, n++) {
      if (¬((n ≡ (l − 1)) ∧ (p→substr(0, (sizeof messageSentinel) − 1) ≡ messageSentinel))) {
        os ≪ *p ≪ endl;
      }
    }
  }
```

This code is used in section 158.

168. Classify message.
The *classifyMessage* class reads input from a **mailFolder** and returns the junk probability for successive messages. The input **mailFolder** may contain only a single message.

⟨ Class definitions 8 ⟩ +≡
```
class classifyMessage {
public:
   mailFolder *mf;
   tokenParser tp;
   unsigned int nExtremal;
   dictionary *d;
   double unknownWordProbability;

   classifyMessage(mailFolder &m, dictionary &dt, unsigned int nExt = 15, double
         uwp = 0.2);

   double classifyThis(void);
protected:
   void addSignificantWordDiagnostics(list⟨string⟩ &l, list⟨string⟩::iterator where,
         multimap⟨double, string⟩ &rtokens);
};
```

169. The constructor initialises the classifier for the default parsing of ISO-8859 messages.

⟨ Global functions 169 ⟩ ≡
```
classifyMessage::classifyMessage(mailFolder &m, dictionary &dt, unsigned int
         nExt = 15, double uwp = 0.2)
{
   mf = &m;
   tp.setSource(m);
   tp.setTokenDefinition(isoToken, asciiToken);
   tp.setTokenLengthLimits(maxTokenLength, minTokenLength, streamMaxTokenLength,
         streamMinTokenLength);
   if (pDiagFilename.length() > 0) {
      tp.setSaveMessage(true);
   }
   d = &dt;
   nExtremal = nExt;
   unknownWordProbability = uwp;
}
```
See also sections 183, 184, 185, and 195.

This code is used in section 204.

170. The *classifyThis* method reads the next message from the mail folder and returns the probability that it is junk. If the end of the mail folder is encountered −1 is returned.

```
⟨ Class implementations 9 ⟩ +≡
  double classifyMessage :: classifyThis(void)
  {
    dictionaryWord dw;
    double junkProb = −1;
    if (transcriptFilename ≠ "") {
      mf→setTranscriptList(&messageTranscript);
      if (Annotate('p') ∨ Annotate('d')) {
        saveParserDiagnostics = true;
      }
    }
    ⟨ Build set of unique tokens in message 172 ⟩;
    ⟨ Classify message tokens by probability of significance 173 ⟩;
    ⟨ Compute probability message is junk from most significant tokens 174 ⟩;
    if (tp.getSaveMessage()) {
      ⟨ Add classification diagnostics to parser diagnostics queue 175 ⟩;
      ofstream mdump(pDiagFilename.c_str());
      tp.writeMessageQueue(mdump);
      mdump.close();
    }
    if (transcriptFilename ≠ "") {
      ⟨ Add annotation to message transcript 176 ⟩;
      mf→writeMessageTranscript(transcriptFilename);
    }
    return junkProb;
  }
```

171. Just one more thing. . . . We need to define an absolute value function for floating point quantities. Make it so.

```
⟨ Class definitions 8 ⟩ +≡
  double abs(double x)
  {
    return (x < 0) ? (−(x)) : x;
  }
```

172. Read the next message from the mail folder and build the **set** *utokens* of unique tokens in the message. **set** insertion automatically discards tokens which appear more than once.

```
⟨ Build set of unique tokens in message 172 ⟩ ≡
  set⟨string⟩ utokens;
  while (tp.nextToken(dw)) {
    utokens.insert(dw.get());
  }
```

This code is used in section 170.

173. Once we've obtained a list of tokens in the message, we now wish to filter it by the significance of the probability that a token appears in junk or legitimate mail. This is simply the absolute value of the difference of the token's *junkProbability* from 0.5—the probability for a token equally likely to appear in junk and legitimate mail. We construct a **multimap** called *rtokens* which maps this significance value to the token string; since any number of tokens may have the same significance, we must use a **multimap** as opposed to a **map**.

We count on **multimap** being an ordered collection class which, when traversed by its **reverse_iterator**, will return tokens in order of significance. This assumption may be unwarranted, but it's valid for all the STL implementations I'm aware of (and is essentially guaranteed since the fact that **multimap** requires only the < operator for ordering effectively mandates a binary tree implementation).

```
⟨ Classify message tokens by probability of significance 173 ⟩ ≡
  multimap⟨double, string⟩ rtokens;
  for (set⟨string⟩ :: iterator t = utokens.begin( ); t ≠ utokens.end( ); t++) {
    double pdiff;
    dictionary :: iterator dp;
    if (((dp = d→find(∗t)) ≠ d→end( )) ∧ (dp→second.getJunkProbability( ) ≥ 0)) {
      pdiff = abs(dp→second.getJunkProbability( ) − 0.5);
    }
    else {
      pdiff = abs(unknownWordProbability − 0.5);
    }
    rtokens.insert(make_pair(pdiff, ∗t));
  }
```

This code is cited in section 206.

This code is used in section 170.

174. Given the list of most signficant tokens, we now use Bayes' theorem to compute the aggregate probability the message is junk. If p_i is the probability word i of the most significant n (`nExtremal`) words in a message appears in junk mail, the probability the message as a whole is junk is:

$$\frac{\prod_{i=1}^{n} p_i}{\prod_{i=1}^{n} p_i + \prod_{i=1}^{n} (1 - p_i)}$$

⟨ Compute probability message is junk from most significant tokens 174 ⟩ ≡

```
unsigned int $n = min(\textbf{static\_cast}\langle\textbf{multimap}\langle\textbf{double},\textbf{string}\rangle :: size\_type\rangle(nExtremal),$
        $rtokens.size());$
multimap⟨double, string⟩ :: $const\_reverse\_iterator\ rp = rtokens.rbegin();$
double $probP = 1,\ probQ = 1;$
if ($verbose$) {
    $cerr \ll$ "Rank␣␣␣Probability␣␣␣Token" $\ll endl;$
}
for (unsigned int $i = 0;\ i < n;\ i{+}{+}$) {
    dictionary :: $iterator\ dp = d\rightarrow find(rp\rightarrow second);$
    double $p = ((dp \equiv d\rightarrow end()) \vee (dp\rightarrow second.getJunkProbability() < 0))\ ?\ unknownWordProbability :$
            $dp\rightarrow second.getJunkProbability();$
    if ($verbose$) {
        $cerr \ll setw(3) \ll setiosflags(ios::right) \ll (i + 1) \ll$ "␣␣␣␣␣␣" $\ll setw(9) \ll setprecision(5) \ll$
            $setiosflags(ios::left) \ll p \ll$ "␣␣" $\ll rp\rightarrow second \ll endl;$
    }
    $probP \mathrel{*}= p;$
    $probQ \mathrel{*}= (1 - p);$
    $rp{+}{+};$
}
$junkProb = probP/(probP + probQ);$
if ($verbose$) {
    $cerr \ll$ "ProbP␣=␣" $\ll probP \ll$ ",␣ProbQ␣=␣" $\ll probQ \ll endl;$
}
```

This code is used in section 170.

175. When parser diagnostics are enabled, add lines to the header of the message in the diagnostic queue to indicate the words we used, their individual probabilities, and the resulting classification of the message as a whole.

```
⟨ Add classification diagnostics to parser diagnostics queue 175 ⟩ ≡
  ostringstream os;
  list⟨string⟩::iterator p;    /* Find the end of the header in the message. If this fails we simply
      append the diagnostics to the end of the message. */
  for (p = tp.messageQueue.begin(); p ≠ tp.messageQueue.end(); p++) {
    if (p→length() ≡ 0) {
      break;
    }
  }
  os ≪ Xfile ≪ "-Junk-Probability:␣" ≪ setprecision(5) ≪ junkProb;
  tp.messageQueue.insert(p, os.str());
  os.str("");
  addSignificantWordDiagnostics(messageTranscript, p, rtokens);
```

This code is used in section 170.

176. If we're producing a message transcript, just before writing it add the annotations to the end of the header which indicate the junk probability and classification of the message based on the threshold settings. After these, other annotations requested by the `--annotate` option are appended.

```
⟨ Add annotation to message transcript 176 ⟩ ≡
  ostringstream os;
  list⟨string⟩ :: iterator p;    /* Find the end of the header in the message. If this fails simply append
      the annotations to the end of the message. */
  for (p = messageTranscript.begin( ); p ≠ messageTranscript.end( ); p++) {
    if (p→length( ) ≡ 0) {
      break;
    }
  }
  double jp = junkProb;    /* If the probability is sufficiently small it to be edited in scientific
      notation, force it to zero so it's easier to parse. */
  if (jp < 0.001) {
    jp = 0;
  }
  os ≪ Xfile ≪ "-Junk-Probability:␣" ≪ setprecision(3) ≪ jp;
  messageTranscript.insert(p, os.str( ));
  os.str("");
  os ≪ Xfile ≪ "-Classification:␣";
  if (junkProb ≥ junkThreshold) {
    os ≪ "Junk";
  }
  else if (junkProb ≤ mailThreshold) {
    os ≪ "Mail";
  }
  else {
    os ≪ "Indeterminate";
  }
  messageTranscript.insert(p, os.str( ));
  if (Annotate('w')) {
    addSignificantWordDiagnostics(messageTranscript, p, rtokens);
  }
  if (Annotate('p') ∨ Annotate('d')) {
    while (¬parserDiagnostics.empty( )) {
      ostringstream os;
      os ≪ Xfile ≪ "-Parser-Diagnostic:␣" ≪ parserDiagnostics.front( );
      messageTranscript.insert(p, os.str( ));
      parserDiagnostics.pop( );
    }
  }
```

This code is used in section 170.

177. Here's the little function which adds the most signficant words and their probabilities to either the parser diagnostics or the transcript. We break it out into a function to avoid duplicating the code.

```
⟨ Class implementations 9 ⟩ +≡
  void classifyMessage :: addSignificantWordDiagnostics(list⟨string⟩ &l,
          list⟨string⟩ :: iterator where, multimap⟨double, string⟩ &rtokens)
  {
    unsigned int n = min(static_cast⟨multimap⟨double, string⟩ :: size_type⟩(nExtremal),
        rtokens.size( ));
    multimap⟨double, string⟩ :: const_reverse_iterator rp = rtokens.rbegin( );
    for (unsigned int i = 0; i < n; i++) {
      dictionary :: iterator dp = d→find(rp→second);
      double wp = ((dp ≡ d→end( )) ∨ ((dp→second.getJunkProbability( ) < 0))) ?
          unknownWordProbability : dp→second.getJunkProbability( );
      ostringstream os;
      os ≪ Xfile ≪ "-Significant-Word:␣" ≪ setw(3) ≪ setiosflags(ios :: right) ≪ (i+1) ≪ "␣␣" ≪
          setw(8) ≪ setprecision(5) ≪ setiosflags(ios :: left) ≪ wp ≪ "␣␣\"" ≪ rp→second ≪ "\"";
      l.insert(where, os.str( ));
      os.str("");
      rp++;
    }
  }
```

178. Main program.
The main program is rather simple. We initialise the global variables then chew through the command line, doing whatever the options request.

```
⟨ Main program 178 ⟩ ≡
  ⟨ Global declarations used by component in temporary jig 203 ⟩;
  int main(int argc, char *argv[])
  {
    int opt;
    ⟨ Initialise global variables 179 ⟩;
    ⟨ Process command-line options 196 ⟩;
    return exitStatus;
  }
```

This code is used in section 204.

179.

```
⟨ Initialise global variables 179 ⟩ ≡
  memset(messageCount, 0, sizeof messageCount);
  isoToken.setISO_8859defaults(minTokenLength, maxTokenLength);
  asciiToken.setUS_ASCIIdefaults(streamMinTokenLength, streamMaxTokenLength);
```

This code is used in section 178.

180. The master dictionary is global to the main program and all of its support functions. It's declared after all the class definitions it requires.

```
⟨ Master dictionary 180 ⟩ ≡
  static dictionary dict;    /* Master dictionary */
```

See also section 199.

This code is used in section 204.

181.

```
⟨ Global variables 181 ⟩ ≡
  static unsigned int messageCount[2];    /* Total messages per category */
  static list⟨string⟩ messageTranscript;    /* Message transcript list */
  static queue⟨string⟩ parserDiagnostics;    /* List of diagnostics generated by the parser */
  static bool saveParserDiagnostics = false;    /* Save parser diagnostics in parserDiagnostics ? */
```

See also sections 194, 200, and 201.

This code is cited in section 194.

This code is used in section 204.

182. The *addFolder* procedure reads a mail folder and adds the tokens it contains to the master dictionary *dict* with the specified *category*. The global *messageCount* for the given *category* is updated to reflect the number of messages added from the folder.

```
⟨ Utility functions 182 ⟩ ≡
  static void addFolder(const char *fname, dictionaryWord::mailCategory cat)
  {
    if (verbose) {
      cerr << "Adding␣folder␣" << fname << "␣as␣" << dictionaryWord::categoryName(cat) <<
          ":" << endl;
    }
    mailFolder mf(fname, cat);
    tokenParser tp;
    tp.setSource(mf);
    tp.setTokenDefinition(isoToken, asciiToken);
    tp.setTokenLengthLimits(maxTokenLength, minTokenLength, streamMaxTokenLength,
        streamMinTokenLength);
    if (pDiagFilename.length() > 0) {
      tp.setSaveMessage(true);
    }
    dictionaryWord dw;
    unsigned int ntokens = 0;
    while (tp.nextToken(dw)) {
      dict.add(dw, mf.getCategory());
      ntokens++;
    }
    messageCount[mf.getCategory()] += mf.getMessageCount();
    if (verbose) {
      cerr << "␣␣Added␣" << mf.getMessageCount() << "␣messages,␣" << ntokens <<
          "␣tokens␣in␣" << mf.getLineCount() << "␣lines." << endl;
      cerr << "␣␣Dictionary␣contains␣" << dict.size() << "␣unique␣tokens." << endl;
    }
  }
```

This code is used in section 204.

183. The *updateProbability* function recomputes word probabilities in the dictionary. It should be called after any changes are made to the contents of the dictionary. Any operation which recomputes the probabilities makes us ineligible for optimising out probability computation loading the first dictionary, so we clear the *singleDictionaryRead* flag.

```
⟨ Global functions 169 ⟩ +≡
  static void updateProbability(void)
  {
    dict.computeJunkProbability(messageCount[dictionaryWord::Mail],
        messageCount[dictionaryWord::Junk], mailBias, minOccurrences);
    singleDictionaryRead = false;
  }
```

184. The *printDictionary* function dumps the dictionary in human-readable form to a specified output stream,

```
⟨ Global functions 169 ⟩ +≡
  static void printDictionary(ostream &os = cout)
  {
    updateProbability();
    os ≪ "Dictionary␣contains␣" ≪ dict.size() ≪ "␣unique␣tokens." ≪ endl;
    for (dictionary::iterator dp = dict.begin(); dp ≠ dict.end(); dp++) {
      dp→second.describe(os);
    }
  }
```

185. The *classifyMessages* function classifies the first message in the mail folder *fname*.

```
⟨ Global functions 169 ⟩ +≡
  static double classifyMessages(const char *fname)
  {
    double jp;
    if (dict.empty()) {
      cerr ≪ "You␣cannot␣--classify␣or␣--test␣a␣message␣""unless␣you␣have␣fir\
          st␣loaded␣a␣dictionary." ≪ endl;
      jp = 0.5;     /* Beats me–call it fifty-fifty junk probability */
    }
    else {
      mailFolder mf(fname, dictionaryWord::Mail);
      classifyMessage cm(mf, dict, significantWords, novelWordProbability);
      jp = cm.classifyThis();
      if (verbose) {
        cerr ≪ "Message␣junk␣probability:␣" ≪ setprecision(5) ≪ jp ≪ endl;
      }
    }
    nTested++;
    return jp;
  }
```

186. Header include files.

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

```
⟨ Include header files 186 ⟩ ≡
#include "config.h"      /* Configuration definitions from ./configure */
  ⟨ Tweak configuration when building for Win32 191 ⟩
  ⟨ C++ standard library include files 187 ⟩
  ⟨ C library include files 188 ⟩
  ⟨ Conditional C library include files 189 ⟩
#include "getopt.h"      /* Use our own getopt, which supports getopt_long */
#include "statlib.h"      /* Statistical library */
  ⟨ Configuration of conditional capabilities 190 ⟩
```

This code is used in section 204.

187. We use the following C++ standard library include files. Note that current C++ theology prescribes that these files not bear the traditional `.h` extension; since some libraries have gotten it into their pointy little heads to natter about this, we conform. If you're using an older C++ system, you may have to restore the `.h` extension if one or more of these come up "not found".

```
⟨ C++ standard library include files 187 ⟩ ≡
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstdlib>
#include <string>
#include <sstream>
#include <vector>
#include <algorithm>
#include <map>
#include <stack>
#include <deque>
#include <queue>
#include <list>
#include <set>
#include <bitset>
#include <functional>
  using namespace std;
```

This code is used in section 186.

188. We also use the following C library include files for low-level operations.

```
⟨ C library include files 188 ⟩ ≡
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <ctype.h>
#include <string.h>
```

This code is used in section 186.

189. Some C library header files are included only on platforms which support the facilities they provide. This is determined by the `./configure` script, which sets variables in `config.h` which we use to include them if present.

⟨ Conditional C library include files 189 ⟩ ≡

```
#ifdef HAVE_STAT
#include <sys/stat.h>
#endif
#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif
#ifdef HAVE_DIRENT_H
#include <dirent.h>
#endif
#ifdef HAVE_MMAP
#include <sys/mman.h>
#include "mystrstream.h"
#endif
```

This code is used in section 186.

190. Some capabilities of the program depend in non-trivial ways on the presence of certain system features detected by the `./configure` script. Here we test for the prerequisites and define an internal tag to enable the feature if all are met.

⟨ Configuration of conditional capabilities 190 ⟩ ≡

```
#if defined (HAVE_GNUPLOT) ∧ defined (HAVE_NETPBM) ∧ defined (HAVE_SYSTEM)
#define HAVE_PLOT_UTILITIES
#endif
#if defined (HAVE_DIRENT_H) ∧ defined (HAVE_STAT)
#define HAVE_DIRECTORY_TRAVERSAL
#endif
#if defined (HAVE_PDFTOTEXT) ∧ defined (HAVE_POPEN) ∧ (defined (HAVE_MKSTEMP) ∨ defined
        (HAVE_TMPNAM))
#define HAVE_PDF_DECODER
#endif
```

This code is used in section 186.

191. It's a pain in the posterior to have to edit the `config.h` file to disable features not supported on Win32 platforms. Since we can't run `./configure` there, the process can't be automated. So, we take the lazy way out and manually undefine features absent on Win32, even if they were auto-detected on the platform which generated `config.h`. Tacky.

⟨ Tweak configuration when building for Win32 191 ⟩ ≡

```
#ifdef WIN32
#undef HAVE_MMAP
#undef HAVE_POPEN
#undef HAVE_DIRENT_H
#endif
```

This code is used in section 186.

192. The following global variables are used to keep track of command line options.

```
#define Annotate(c)  (annotations.test(c))    /* Test if annotation is requested */
⟨ Command line arguments 192 ⟩ ≡
  static double mailBias = 2.0;    /* Bias for words in legitimate mail */
  static unsigned int minOccurrences = 5;    /* Minimum occurrences to trust probability */
  static double junkThreshold = 0.9;    /* Threshold above which we classify mail as junk */
  static double mailThreshold = 0.9;    /* Threshold below which we classify as mail */
  static int significantWords = 15;    /* Number of words to use in classifying message */
  static double novelWordProbability = 0.2;
    /* Probability assigned to words not in dictionary */
  static bitset⟨1 ≪ (sizeof(char) * 8)⟩ annotations;    /* Annotations requested in transcript */
```

See also section 193.

This code is used in section 204.

193. These globals are used to check for inconsistent option specifications.

```
⟨ Command line arguments 192 ⟩ +≡
  static unsigned int nTested = 0;    /* Number of messages tested */
```

194. The following options are referenced in class definitions and must be placed in the ⟨ Global variables 181 ⟩ section so they'll be declared prior to references to them.

```
⟨ Global variables 181 ⟩ +≡
  static bool verbose = false;    /* Print verbose processing information */
#ifdef TYPE_LOG
  static ofstream typeLog("/tmp/typelog.txt");
#endif
  static string pDiagFilename = "";    /* Parser diagnostic file name */
  static string transcriptFilename = "";    /* Message transcript file name */
  static bool pTokenTrace = false;    /* Include detailed token trace in pDiagFilename output ? */
  static unsigned int maxTokenLength = 64, minTokenLength = 1;
    /* Minimum and maximum token length limits */
  static unsigned int streamMaxTokenLength = 64, streamMinTokenLength = 5;
    /* Minimum and maximum byte stream token length limits */
  static bool singleDictionaryRead = true;
    /* Can we optimise probability computation after dictionary import ? */
  static unsigned int phraseMin = 1, phraseMax = 1;
    /* Minimum and maximum phrase length in words */
  static unsigned int phraseLimit = 0;    /* Maximum phrase length */
```

195. 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 169 ⟩ +≡
  static void usage(void)
  {
    cout ≪ PRODUCT ≪ "␣␣--␣␣Annoyance␣Filter.␣␣Call" ≪ endl;
    cout ≪ "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣with␣" ≪ PRODUCT ≪ "␣[options]" ≪ endl;
    cout ≪ "" ≪ endl;
    cout ≪ "Options:" ≪ endl;
    cout ≪ "␣␣␣␣--annotate␣options␣␣␣␣␣Specify␣optional␣annotations␣in␣--transcript" ≪
        endl;
    cout ≪ "␣␣␣␣--binword␣n␣␣␣␣␣␣␣␣␣␣␣␣Scan␣binary␣streams␣for␣words␣>=\
        ␣n␣characters␣(0␣=␣none)" ≪ endl;
    cout ≪ "␣␣␣␣--classify␣fname␣␣␣␣␣␣␣Classify␣first␣message␣in␣fname" ≪ endl;
    cout ≪ "␣␣␣␣--clearjunk␣␣␣␣␣␣␣␣␣␣␣␣Clear␣junk␣counts␣in␣dictionary" ≪ endl;
    cout ≪ "␣␣␣␣--clearmail␣␣␣␣␣␣␣␣␣␣␣␣Clear␣mail␣counts␣in␣dictionary" ≪ endl;
    cout ≪ "␣␣␣␣--copyright␣␣␣␣␣␣␣␣␣␣␣␣Print␣copyright␣information" ≪ endl;
    cout ≪ "␣␣␣␣--csvread␣fname␣␣␣␣␣␣␣␣Import␣dictionary␣from␣fname␣in␣CSV␣format" ≪
        endl;
    cout ≪ "␣␣␣␣--csvwrite␣fname␣␣␣␣␣␣␣Export␣dictionary␣to␣fname␣in␣CSV␣format" ≪ endl;
    cout ≪ "␣␣␣␣--help,␣-u␣␣␣␣␣␣␣␣␣␣␣␣␣Print␣this␣message" ≪ endl;
#ifdef Jig
    cout ≪ "␣␣␣␣--jig␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Test␣component␣in␣temporary␣jig" ≪ endl;
#endif
    cout ≪ "␣␣␣␣--junk,␣-j␣folder␣␣␣␣␣␣Add␣folder␣contents␣to␣junk␣mail␣dictionary" ≪
        endl;
    cout ≪ "␣␣␣␣--list␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Print␣dictionary␣on␣standard␣output" ≪ endl;
    cout ≪ "␣␣␣␣--mail,␣-m␣folder␣␣␣␣␣␣Add␣folder␣contents␣to␣legitimat\
        e␣mail␣dictionary" ≪ endl;
    cout ≪ "␣␣␣␣--newword␣n␣␣␣␣␣␣␣␣␣␣␣␣Set␣probability␣for␣words␣not␣in\
        ␣dictionary␣to␣n" ≪ endl;
    cout ≪ "␣␣␣␣--pdiag␣fname␣␣␣␣␣␣␣␣␣␣Print␣parser␣diagnostics␣to␣fname" ≪ endl;
    cout ≪ "␣␣␣␣--phraselimit␣n␣␣␣␣␣␣␣␣Set␣phrase␣maximum␣length␣to␣n␣characters" ≪
        endl;
    cout ≪ "␣␣␣␣--phrasemax␣n␣␣␣␣␣␣␣␣␣␣Set␣phrase␣maximum␣to␣n␣words" ≪ endl;
    cout ≪ "␣␣␣␣--phrasemin␣n␣␣␣␣␣␣␣␣␣␣Set␣phrase␣minimum␣to␣n␣words" ≪ endl;
#ifdef HAVE_PLOT_UTILITIES
    cout ≪ "␣␣␣␣--plot␣fname␣␣␣␣␣␣␣␣␣␣␣Plot␣histogram␣of␣word␣probabili\
        ties␣in␣dictionary" ≪ endl;
#endif
    cout ≪ "␣␣␣␣--prune␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Prune␣infrequently␣used␣words␣from␣dictionary" ≪
        endl;
    cout ≪ "␣␣␣␣--ptrace␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Include␣detailed␣trace␣in␣--pdiag␣output" ≪ endl;
    cout ≪ "␣␣␣␣--read,␣-r␣fname␣␣␣␣␣␣␣Import␣dictionary␣from␣fname" ≪ endl;
    cout ≪ "␣␣␣␣--sigwords␣n␣␣␣␣␣␣␣␣␣␣␣Classify␣message␣based␣on␣n␣most\
        ␣significant␣words" ≪ endl;
    cout ≪ "␣␣␣␣--statistics␣␣␣␣␣␣␣␣␣␣␣Print␣statistics␣of␣dictionary" ≪ endl;
    cout ≪ "␣␣␣␣--test,␣-t␣fname␣␣␣␣␣␣␣Test␣first␣message␣in␣fname" ≪ endl;
    cout ≪ "␣␣␣␣--threshjunk␣n␣␣␣␣␣␣␣␣␣Set␣junk␣threshold␣to␣n" ≪ endl;
    cout ≪ "␣␣␣␣--threshmail␣n␣␣␣␣␣␣␣␣␣Set␣mail␣threshold␣to␣n" ≪ endl;
    cout ≪ "␣␣␣␣--transcript␣fname␣␣␣␣␣Write␣annotated␣message␣transcript␣to␣fname" ≪
        endl;
    cout ≪ "␣␣␣␣--verbose,␣-v␣␣␣␣␣␣␣␣␣␣Print␣processing␣information" ≪ endl;
    cout ≪ "␣␣␣␣--version␣␣␣␣␣␣␣␣␣␣␣␣␣␣Print␣version␣number" ≪ endl;
    cout ≪ "␣␣␣␣--write␣fname␣␣␣␣␣␣␣␣␣␣Export␣dictionary␣to␣fname" ≪ endl;
    cout ≪ "" ≪ endl;
    cout ≪ "by␣John␣Walker" ≪ endl;
    cout ≪ "http://www.fourmilab.ch/" ≪ endl;
  }
```

196. We use *getopt_long* to process command line options. This permits aggregation of single letter options without arguments and both `-d`*arg* 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 196 ⟩ ≡
  static const struct option long_options[] = {
  {"annotate", 1, Λ, 222},
  {"binword", 1, Λ, 221},
  {"classify", 1, Λ, 209},
  {"clearjunk", 0, Λ, 215},
  {"clearmail", 0, Λ, 216},
  {"copyright", 0, Λ, 200},
  {"csvread", 1, Λ, 205},
  {"csvwrite", 1, Λ, 207},
  {"help", 0, Λ, 'u'},
#ifdef Jig
  {"jig", 0, Λ, 206},
#endif
  {"junk", 1, Λ, 'j'},
  {"list", 0, Λ, 202},
  {"mail", 1, Λ, 'm'},
  {"newword", 1, Λ, 220},
  {"pdiag", 1, Λ, 212},
  {"phraselimit", 1, Λ, 224},
  {"phrasemax", 1, Λ, 223},
  {"phrasemin", 1, Λ, 217},
#ifdef HAVE_PLOT_UTILITIES
  {"plot", 1, Λ, 211},
#endif
  {"prune", 0, Λ, 203},
  {"ptrace", 0, Λ, 213},
  {"purge", 0, Λ, 203},     /* For compatibility, it's --prune now */
  {"read", 1, Λ, 'r'},
  {"sigwords", 1, Λ, 219},
  {"statistics", 0, Λ, 210},
  {"test", 1, Λ, 't'},
  {"threshjunk", 1, Λ, 208},
  {"threshmail", 1, Λ, 214},
  {"transcript", 1, Λ, 204},
  {"verbose", 0, Λ, 'v'},
  {"version", 0, Λ, 201},
  {"write", 1, Λ, 218},
  {0, 0, 0, 0}
  };
  int option_index = 0;
  bool lastOption = false;     /* Set true to exit command line processing after option */
  int exitStatus = 0;     /* Program exit status */
  while ((¬lastOption) ∧ (opt = getopt_long(argc, argv, "j:m:r:t:uv", long_options,
        &option_index)) ≠ −1) {
    switch (opt) {
    case 222:     /* --annotate options Add annotation options to --transcript output */
      while ((*optarg) ≠ 0) {
        unsigned int ch = (*optarg++) & #FF;
        if (isascii(ch) ∧ isalpha(ch) ∧ isupper(ch)) {
          ch = islower(ch);
        }
        annotations.set(ch);
      }
      break;
    case 221:     /* --binwords n Parse binary streams for words of n characters or more */
      streamMinTokenLength = atoi(optarg);
      if (verbose) {
        if (streamMinTokenLength > 0) {
          cerr ≪ "Binary␣streams␣will␣be␣parsed␣for␣words␣of␣" ≪ streamMinTokenLength ≪
              "␣characters␣or␣more." ≪ endl;
        }
        else {
          cerr ≪ "Binary␣streams␣will␣not␣be␣parsed␣for␣words." ≪ endl;
        }
      }
      break;
    case 209:     /* --classify fname Classify message in fname */
      {
        if (optind < argc) {
          cerr ≪ "Warning:␣command␣line␣arguments␣after␣\"--classify␣" ≪ optarg ≪
              "␣will␣be␣ignored." ≪ endl;
        }
        double score = classifyMessages(optarg);
        if (score ≥ junkThreshold) {
          cout ≪ "JUNK" ≪ endl;
          exitStatus = 3;
        }
        else if (score ≤ mailThreshold) {
          cout ≪ "MAIL" ≪ endl;
          exitStatus = 0;
        }
        else {
          cout ≪ "INDT" ≪ endl;     /* "INDeTerminate" */
          exitStatus = 4;
        }
        lastOption = true;     /* Bail out, ignoring any (erroneous) subsequent options */
        break;
      }
    case 215:     /* --clearjunk Clear junk counts in dictionary */
      dict.resetCat(dictionaryWord :: Junk);
      messageCount[dictionaryWord :: Junk] = 0;
      break;
    case 216:     /* --clearmail Clear mail counts in dictionary */
      dict.resetCat(dictionaryWord :: Mail);
      messageCount[dictionaryWord :: Mail] = 0;
      break;
    case 200:    /* --copyright Print copyright information */
      cout ≪ "This␣program␣is␣in␣the␣public␣domain.\n";
      return 0;
    case 205:    /* --csvead fname Import dictionary from CSV fname */
      {
        ifstream is(optarg);
        if (¬is) {
          cerr ≪ "Cannot␣open␣CSV␣dictionary␣file␣" ≪ optarg ≪ endl;
          return 1;
        }
        dict.importCSV(is);
        if (¬singleDictionaryRead) {
          updateProbability( );
        }
        singleDictionaryRead = false;
        is.close( );
      }
      break;
    case 207:    /* --csvwrite fname Export dictionary to CSV fname */
      {
        ofstream of(optarg);
        if (¬of) {
          cerr ≪ "Cannot␣create␣CSV␣export␣file␣" ≪ optarg ≪ endl;
          return 1;
        }
        updateProbability( );
        dict.exportCSV(of);
        of.close( );
      }
      break;
    case 'u':    /* -u, --help Print how-to-call information */
    case '?':    /* -? Indication of error parsing command line */
      usage( );
      return 0;
#ifdef Jig
    case 206:    /* --jig Test component in temporary jig */
      {
        ⟨Test component in temporary jig 202⟩;
      }
      break;
#endif
    case 'j':    /* -j, --junk folder Add folder contents to junk mail dictionary */
      addFolder(optarg, dictionaryWord::Junk);
      updateProbability( );
      break;
    case 202:    /* --list Print dictionary on standard output */
      printDictionary( );
      break;
    case 'm':    /* -m, --mail folder Add folder contents to legitimate mail dictionary */
      addFolder(optarg, dictionaryWord::Mail);
```

```
        updateProbability( );
        break;
    case 220:   /* --newword n Set probability for words not in dictionary to n */
        novelWordProbability = atof(optarg);
        if (verbose) {
          cerr ≪ "Probability␣for␣words␣not␣in␣dictionary␣set␣to␣" ≪ novelWordProbability ≪
              "." ≪ endl;
        }
        break;
    case 212:   /* --pdiag fname Write parser diagnostic log to fname */
        pDiagFilename = optarg;
        break;
    case 224:   /* --phraselimit n Set phrase maximum length to n characters */
        phraseLimit = atoi(optarg);
        if (verbose) {
          cerr ≪ "Phrase␣maximum␣length␣set␣to␣" ≪ phraseLimit ≪ "␣characters." ≪ endl;
        }
        break;
    case 223:   /* --phrasemax n Set phrase maximum to n words */
        phraseMax = atoi(optarg);
        if (verbose) {
          cerr ≪ "Phrase␣maximum␣length␣set␣to␣" ≪ phraseMax ≪ "␣word" ≪ (phraseMax ≡ 1 ?
              "" : "s") ≪ "." ≪ endl;
        }
        break;
    case 217:   /* --phrasemin n Set phrase minimum to n words */
        phraseMin = atoi(optarg);
        if (verbose) {
          cerr ≪ "Phrase␣minimum␣length␣set␣to␣" ≪ phraseMin ≪ "␣word" ≪ (phraseMin ≡ 1 ?
              "" : "s") ≪ "." ≪ endl;
        }
        break;
#ifdef HAVE_PLOT_UTILITIES
    case 211:   /* --plot fname Plot dictionary histogram as fname.png */
        updateProbability( );
        dict.plotProbabilityHistogram(optarg);
        break;
#endif
    case 203:   /* --prune Purge dictionary of infrequently used words */
        updateProbability( );
        dict.purge( );
        break;
    case 213:   /* --ptrace Include token by token trace in --pdiag output */
        pTokenTrace = true;
        break;
    case 'r':   /* -r, --read fname Read dictionary from fname */
        {
#ifdef HAVE_MMAP
          int fileHandle = open(optarg, O_RDONLY);
          if (fileHandle ≡ −1) {
            cerr ≪ "Cannot␣open␣dictionary␣file␣" ≪ optarg ≪ endl;
            return 1;
          }
          long fileLength = lseek(fileHandle, 0, 2);
          lseek(fileHandle, 0, 0);
          char *dp = static_cast⟨char *⟩(mmap((caddr_t)0, fileLength, PROT_READ,
              MAP_SHARED | MAP_NORESERVE, fileHandle, 0));
          istrstream is(dp, fileLength);
#else
          ifstream is(optarg, ios::binary);
          if (¬is) {
            cerr ≪ "Cannot␣open␣dictionary␣file␣" ≪ optarg ≪ endl;
            return 1;
          }
#endif
          dict.importFromBinaryFile(is);
#ifdef HAVE_MMAP
          munmap(dp, fileLength);
          close(fileHandle);
#else
          is.close( );
#endif
          if (¬singleDictionaryRead) {
            updateProbability( );
          }
          singleDictionaryRead = false;
        }
        break;
    case 219:    /* --sigwords n Classify message based on n most significant words */
        significantWords = atoi(optarg);
        if (verbose) {
          cerr ≪ "Significant␣words␣set␣to␣" ≪ significantWords ≪ "." ≪ endl;
        }
        break;
    case 210:    /* --statistics Print statistics of dictionary */
        updateProbability( );
        dict.printStatistics( );
        break;
    case 't':    /* -t, --test fname Test message in fname */
        {
          double score = classifyMessages(optarg);
          if (transcriptFilename ≠ "-") {
            cout ≪ "Junk␣probability␣" ≪ score ≪ endl;
          }
        }
        break;
    case 208:    /* --threshjunk n Set junk threshold to n */
        junkThreshold = atof(optarg);
        if (verbose) {
```

```
            cerr ≪ "Junk␣threshold␣set␣to␣" ≪ setprecision(5) ≪ junkThreshold ≪ "." ≪ endl;
         }
         break;
      case 214:    /* --threshmail n Set mail threshold to n */
         mailThreshold = atof(optarg);
         if (verbose) {
            cerr ≪ "Mail␣threshold␣set␣to␣" ≪ setprecision(5) ≪ mailThreshold ≪ "." ≪ endl;
         }
         break;
      case 204:    /* --transcript fname Write annotated message transcript to fname */
         transcriptFilename = optarg;
         break;
      case 'v':    /* -v, --verbose Print processing information */
         verbose = true;
         break;
      case 201:    /* --version Print version information */
         cout ≪ PRODUCT "␣" VERSION ≪ endl;
         cout ≪ "Last␣revised:␣" REVDATE ≪ endl;
         cout ≪ "The␣latest␣version␣is␣always␣available␣from:" ≪ endl;
         cout ≪ "␣␣␣␣http://www.fourmilab.ch/annoyance-filter/" ≪ endl;
         cout ≪ "Please␣report␣bugs␣to:" ≪ endl;
         cout ≪ "␣␣␣␣bugs@fourmilab.ch" ≪ endl;
         return 0;
      case 218:    /* --write fname Write dictionary to fname */
         {
            ofstream of(optarg, ios::binary);
            if (¬of) {
               cerr ≪ "Cannot␣create␣dictionary␣file␣" ≪ optarg ≪ endl;
               return 1;
            }
            updateProbability();
            dict.exportToBinaryFile(of);
            of.close();
         }
         break;
      default:
         cerr ≪ "***Internal␣error:␣unhandled␣case␣" ≪ opt ≪ "␣in␣option␣processing." ≪
               endl;
         return 1;
      }
   }
   ⟨ Check for inconsistencies in option specifications 197 ⟩;
```

This code is used in section 178.

197. Some combinations of option specifications make no sense or indicate the user doesn't understand how they're processed. Check for such circumstances and issue warnings to point out the error of the user's ways.

```
⟨ Check for inconsistencies in option specifications 197 ⟩ ≡
  if (pTokenTrace ∧ (pDiagFilename ≡ "")) {
    cerr ≪ "Warning:␣--ptrace␣requested␣but␣no␣--pdiag␣file␣specified." ≪ endl;
  }
  if ((transcriptFilename ≠ "") ∧ (nTested ≡ 0)) {
    cerr ≪ "Warning:␣--transcript␣requested␣but␣no␣message␣--test␣or␣--\
        classify␣done." ≪ endl;
  }
  if ((pDiagFilename ≠ "") ∧ (nTested ≡ 0)) {
    cerr ≪ "Warning:␣--pdiag␣requested␣but␣no␣message␣--test␣or␣--classify␣done." ≪
        endl;
  }
  if (annotations.count() > 0 ∧ (transcriptFilename ≡ "")) {
    cerr ≪ "Warning:␣--annotate␣requested␣but␣no␣--transcript␣file␣specified." ≪ endl;
  }
```

This code is used in section 196.

198. 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.

199. Define the various kinds of tokens we parse from the input stream.

```
⟨ Master dictionary 180 ⟩ +≡
  static tokenDefinition isoToken;    /* ISO-8859 token definition */
  static tokenDefinition asciiToken;    /* US-ASCII token definition */
```

200. ISO 8859-1 character types.

The following definitions provide equivalents for `ctype.h` macros which work for ISO-8859 8 bit characters. They require that `ctype.h` be included before they're used.

⟨ Global variables 181 ⟩ +≡
#define $\mathit{ISOch}(x)$ (**static_cast**⟨**unsigned char**⟩$((x)\ \&\ {}^\#\mathtt{FF}))$
#define $\mathit{isISOspace}(x)$ $(\mathit{isascii}(\mathit{ISOch}(x)) \land \mathit{isspace}(\mathit{ISOch}(x)))$
#define $\mathit{isISOalpha}(x)$ $((\mathit{isoalpha}[\mathit{ISOch}(x)/8]\ \&\ ({}^\#\mathtt{80} \gg (\mathit{ISOch}(x)\ \%\ 8))) \neq 0)$
#define $\mathit{isISOupper}(x)$ $((\mathit{isoupper}[\mathit{ISOch}(x)/8]\ \&\ ({}^\#\mathtt{80} \gg (\mathit{ISOch}(x)\ \%\ 8))) \neq 0)$
#define $\mathit{isISOlower}(x)$ $((\mathit{isolower}[\mathit{ISOch}(x)/8]\ \&\ ({}^\#\mathtt{80} \gg (\mathit{ISOch}(x)\ \%\ 8))) \neq 0)$
#define $\mathit{toISOupper}(x)$ $(\mathit{isISOlower}(x)\ ?\ (\mathit{isascii}((($**unsigned**
char$)(x)))\ ?\ \mathit{toupper}(x) : (((\mathit{ISOch}(x) \neq {}^\#\mathtt{DF}) \land (\mathit{ISOch}(x) \neq {}^\#\mathtt{FF}))\ ?\ (\mathit{ISOch}(x) - {}^\#\mathtt{20}) : (x))) : (x))$
#define $\mathit{toISOlower}(x)$ $(\mathit{isISOupper}(x)\ ?\ (\mathit{isascii}(\mathit{ISOch}(x))\ ?\ \mathit{tolower}(x) : (\mathit{ISOch}(x) + {}^\#\mathtt{20})) : (x))$

201. The following tables are bit vectors which define membership in the character classes tested for by the preceding macros.

⟨ Global variables 181 ⟩ +≡
const unsigned char $\mathit{isoalpha}[32] = \{0,0,0,0,0,0,0,0,127,255,255,224,127,255,255,224,0,0,0,0,0,0,0,0,255,255,254,255,255,255,254,255\}$;
const unsigned char $\mathit{isoupper}[32] = \{0,0,0,0,0,0,0,0,127,255,255,224,0,0,0,0,0,0,0,0,0,0,0,0,255,255,254,254,0,0,0,0\}$;
const unsigned char $\mathit{isolower}[32] = \{0,0,0,0,0,0,0,0,0,0,0,0,127,255,255,224,0,0,0,0,0,0,0,0,0,0,0,1,255,255,254,255\}$;

202. To perform component tests during the development process we provide a *test jig* in which the component may be figuratively mounted and exercised. When compiled with `Jig` defined, a `--jig` option (without argument) is included to activate the test.

⟨ Test component in temporary jig 202 ⟩ ≡
#ifdef *Jig*
#endif

This code is used in section 196.

203. The component in the temporary test jig may require some items declared in global context. Here's where you can put such declarations.

⟨ Global declarations used by component in temporary jig 203 ⟩ ≡
#ifdef *Jig*
#endif

This code is used in section 178.

204. Overall program structure.

Here we put all the pieces together in the order required by the digestive tract of the C++ compiler. Like programmers, who must balance their diet among the four basic food groups: sugar, salt, fat, and caffeine, compilers require a suitable mix of definitions, declarations, classes, and functions to get along. Compilers are rather more picky than programmers in the order in which these delectations are consumed.

⟨ Preprocessor definitions ⟩
⟨ Include header files 186 ⟩
⟨ Global variables 181 ⟩
⟨ Class definitions 8 ⟩
⟨ Command line arguments 192 ⟩
⟨ Class implementations 9 ⟩
⟨ Master dictionary 180 ⟩
⟨ Global functions 169 ⟩
⟨ Utility functions 182 ⟩
⟨ Main program 178 ⟩

205. Release history.

Release 0.1: October 2002

Initial release.

206. Development log.

2002 August 28

Created development tree and commenced implementation.

2002 September 1

Release 0.1 circulated for review.

2002 September 6

Added the ability to compute descriptive statistics of the dictionary built by parsing the `--mail` and `--junk` folders, using the facilities of the `statlib.w` program. Statistics are written to standard output.

Added a `--plot` option to plot a histogram of words in a newly parsed dictionary (not a lookup dictionary loaded with `--read`). Creating the plot requires the `GNUPLOT` and `PBMPlus` utilities to be installed.

2002 September 7

Well, after a huge amount of hunkering down and twiddling, parsing of MIME multi-part messages and decoding of parts encoded in `Base64` and `Quoted-Printable` encoding now seems to be working. This drastically improves the quality of parsing, particularly for junk where these forms of encoding are used as "stealth" to evade other content-based filters.

2002 September 8

Added the ability to read mail folders compressed with `gzip` or other compressors detected by the Autoconf script. This saves a lot of space when you're keeping large training archives around. This will work only on systems with suitable decompressors and the *popen* facility.

2002 September 9

Added the `--pdiag` option to write the parser diagnostics to a designated file. Previously this was controlled by a gnarly # **define**.

Added a "`X-Annoyance-Filter-Decoder`" line to the `--pdiag` output to indicate the activation of decoders (including the sink) for MIME parts in the message. These lines are not seen by the token parser.

Fixed a bug in parsing of tokens including ISO accented characters. . .signed characters strike again.

2002 September 10

Added a `--ptrace` option to include the actual tokens parsed as indented, quoted lines following each line of parser input in the `--pdiag` file.

Added code to **classifyMessage** which appends lines to the message header in the `--pdiag` file giving the aggregate junk probability and the most significant words and their individual probabilities.

Separated the mail and junk thresholds, which may now be set independently by the `--threshjunk` and `--threshmail` options. The `--classify` command now writes "`INDT`" (for "indeterminate") if a message falls between the two thresholds and exits with a return status of 4.

Added the `--binwrite` and `--binread` options to export and import a **dictionary** as a portable (assuming IEEE floating point on all platforms) binary file. This will permit easier distribution of dictionary databases and may be faster to load than the *lookupDictionary*.

Added the `--clearjunk` and `--clearmail` options to clear counts of junk and mail. This can be used, in conjunction with the `binwrite` option, to prepare databases for use by folks who do not wish to prepare their own.

2002 September 11

Added the ability to enforce minimum and maximum length constraints on tokens returned by **tokenParser**. The limits are set to accept tokens from 1 to 255 characters in the **tokenParser** constructor, and may be changed at any time with the *setTokenLengthLimits* method. Note that the length limits are not reset by a call to *setSource*.

Set the default token parser length limits to accept tokens between 1 and 64 characters. This will doubtless be the subject of yet more command line options before long.

Modified the code which decides whether a mail folder is compressed to check for the argument being a symbolic link. If so, the link target is tested for the extension indicating a compressed file. I only follow links one level—if this poses a problem, your life is probably too complicated.

Fixed computation of probability to avoid crashes if no words are present in a category. Probabilities don't make any sense in such circumstances, but you may wish to create such a database for use with `--binread`.

Added logic to **dictionary** :: *exportToBinaryFile* and **dictionary** :: *exportToBinaryFile* to save and restore the count of messages contributing to the dictionary in the *messageCount* array in a pseudo-word called "␣COUNTS␣" (obligatorily) at the start of the dictionary. These counts are required should we need to recompute the probability subsequent to loading the dictionary.

Added the `--newword` and.–sigwords options to specify the probability given to words in a message which don't appear in the dictionary and the number of "most significant" words whose probabilities are used to determine the aggregate probability a given message is junk.

2002 September 12

Added logic to cope with the body of a message being encoded in a `Content-TransferEncoding`. While processing the header, this and the `Content-Type` are parsed as in MIME headers, with their arguments saved in *bodyContentType*, *bodyContentTypeCharset*, and *bodyContentTransferEncoding*. At the end of the header, if a *bodyContentTransferEncoding* has been specified, the values are transferred to the corresponding *mime*... variables and *multiPart* is set with an end terminator of the null string. The latter disables the decoder's test for a part end sentinel and the warning for an unterminated part.

Messages with `Subject` lines which contain ISO 8859 encoded characters employ a form of `Quoted-Printable` encoding to permit these characters to appear in a mail header where only 7 bit ASCII is permitted. I added code to **mailFolder** to detect these lines and call a new *decodeEscapedText* method of **quotedPrintableMIMEdecoder** to decode them if properly formed. This will permit parsing of ISO subject lines, which may prove critical in discriminating among messages with very short body copy.

Yikes! As far as I can determine from the RFCs, what we're supposed to do with continued header lines is just concatenate them, discarding all white space on the continuation even if this runs together tokens on adjacent lines. At least, if you don't do this, encoded words split across continued `Subject` lines end up with nugatory white space in the middle. So, I fixed ⟨ Check for continuation of mail header lines 132 ⟩ to "work this way". Given our definition of tokens, it's likely to fix more things than it breaks anyway.

Added documentation to the `CWEB` file for yesterday's new options.

2002 September 13

`Subject` lines can, of course, also contain sequences encoded in `Base64`, tagged with a “`?B?`” following the *charset* specification. Added decoding of these sequences, along with the requisite *decodeEscapedText* method of **base64MIMEdecoder**.

Made a slight revision to the definition of tokens in the **tokenParser**. While “`-`” and “`'`” continue to be considered part of a token if embedded within it, they can no longer be the first or last characters of a token. This improves recognition of words in typical text, based on tests against the big collection. A new *not_at_ends* array of **bool** is used to define which characters may not begin or end a token.

Completely rewrote how the **tokenParser** determines character types in parsing for tokens. Previously, characters were classified by looking them up in a collection of global arrays of **bool**. To permit changing the definition of a token on the fly, I defined a new class, **tokenDefinition**, which collects together the lookup tables which determine which characters constitute a token and indicate the sets of characters (if any) which cannot exclusively make up a token and which cannot be the first or last character of a token. In addition, the minimum and maximum acceptable length for tokens are stored and methods permit testing all of these quantities. You can initialise the values as you with the methods provided, or use pre-defined initialiser functions for ISO-8859 and ASCII alphanumeric sets.

Well, let’s declare this a red banner day for the `annoyance-filter`! No, you’re not dreaming…we’re actually ending this day with *fewer* command line options than those which greeted the dawn, and the whole concept of the “lookup dictionary” has been banished, along with snowdrifts of prose in the documentation explaining the difference between a “dictionary” and a ‘lookup dictionary” and the things you could or couldn’t do with, or to, them respectively. The original idea was that you work with **dictionary** objects when assembling the database of mail and junk, and then export the results as a lean and mean lookup dictionary which could be loaded like lightning to classify subsequent messages. Well, it turns out that if you use binary I/O for the **dictionary**, it’s just as fast as loading the lookup dictionary, and all of the confusion is eliminated. Further, the user is thereby encouraged to keep a dictionary on hand which can be updated at any time to incorporate new examples of mail and junk. This is all much more the Bayesian spirit of eternal refinement than settling on a probability set without subsequent refinement.

Since the lookup dictionary is no more, there’s no need to distinguish the **dictionary** read and write commands as binary. Hence, the `--binread` and `--binwrite` options have been renamed `--read` and `--write`, freed up by the lookup dictionary elimination.

2002 September 14

The direct concatenation of multiple-line header items added a couple of days ago broke ⟨ Process multipart MIME header declaration 137 ⟩ thanks to fat-fingered character counting in the recognition of sentinels. I fixed this, and modified the code to perform all parsing on a canonicalised string to avoid case sensitivity problems. Note that the `boundary` itself *is* and must remain case sensitive.

Fixed some `gcc -Wall` natters which had crept in since the option was accidentally removed by `autoconf`.

Added the ability to read a **mailFolder** from standard input. If the *fname* argument to the constructor is “`-`” *cin* is used as the input stream.

Renamed the `--csv` option `--csvwrite` in keeping with nefarious plans soon to be disclosed, and added a pseudo “`␣COUNTS␣`” word to the start of the CSV file giving the number of mail and junk messages in the dictionary as is done in binary dictionary dumps. Changed the sort order for the CSV file so that words with identical probabilities are sorted into lexical order.

Added a `--csvread` option to import a dictionary from a CSV file in the format created by `--csvwrite`. The CSV file is *added* to the existing in-memory dictionary; multiple `--csvread` and `--read` command may be used to assemble a dictionary. The CSV file imported need not be sorted in any particular order and may contain comments whose first nonblank character is “`;`” or “`#`”. In the process, I found and fixed a bug in updating the message counts which applied to both `--csvread` and the existing `--read` code, but which only manifested itself when loading multiple dictionaries.

Wheels within wheels. . .MIME `multipart` messages can, of course, be nested. You can be blithely parsing your way through a message when you trip over a part with a `Content-type` of "`multipart/alternative`", which pushes a new part boundary onto the stack, to be popped when the end sentinel of that nested section is encountered. What fun. We consequently introduce a new *partBoundaryStack* to keep track of the nested part boundary sentinels, along with all of the defensive code needed to cope with the realities of real world mail.

2002 September 15

Loosened up the test for `multipart Content-type` so that "`multipart/related`" types will be recognised.

Added the long-awaited `--transcript` option. (Thanks, Kern, for suggesting it!) A transcript of the input message for a `--test` or `--classify` operation is written to the argument file name (standard output if the argument is "`-`", with `X-Annoyance-Filter-Junk-Probability` and `X-Annoyance-Filter-Classification` items appended to the header indicating the calculated junk probability and classification according to the thresholds.

Finished the first cut of multiple byte character set decoders and interpreters. A *decoder* scans the mail body (encoded or not), and parses the byte stream into logical characters up to 32 bits in width. An *interpreter* expresses these characters in a form suitable for analysis. Ideographic languages are typically interpreted as one word per character, other languages as one letter per character. These components must, of course, be utterly bullet-proof as they will be subjected to every possibly kind of garbage in the course of parsing real-world mail. At the moment, we have decoders for EUC and Big5, and interpreters for GB2312 and Big5.

Added a decoder for EUC-encoded Korean (`euc-kr`) as an example of how to handle an alphabetic language with a non-Western character set.

2002 September 16

Modified **EUC_MBCSdecoder** to discard the balance of any encoded line in which an invalid EUC second byte is encountered. After encountering such garbage, the rest of the line is usually junk and there's no profit in blithering through it.

Added logic to scan `application` binary byte streams for possible embedded tokens. The new `--binword` option sets the shortest sequence of contiguous ASCII alphanumeric characters or dollar signs (with possible embedded hyphens and apostrophes, but not permitting these character at the start or end of a token—the default is 5 characters, which is a tad more discriminating than the UNIX `strings` which defaults to 4 printable characters. You can disable the scanning of binary streams entirely by setting `--binword` to zero. Scanning binary streams might seem to be a curious endeavour, but it's highly effective at percolating text embedded in viruses and worm attachments to junk mail to the top of the junk probability hit parade, then screening them out when the arrive in incoming mail.

Although the `Subject` line is the most important, any line in a mail header may actually contain quoted sequences specifying a character set and `Quoted-Printable` or `Base64` encoded characters. I modified ⟨ Check for encoded header line and decode 135 ⟩ to no longer restrict decoding to the subject line.

Once decoded, if the `charset` specification in a header line quoted sequence is a character set we understand, it is not decoded and interpreted. `ISO-8859` sets of all flavours are decoded but not processed further.

Fixed a few `gcc -Wall` quibbles in **tokenDefinition** which popped up on Solaris compiler but didn't seem to perturb the almost identical version of `gcc` on Linux.

Modified the `--test` option so that if the `--transcript` option has been previously specified with standard output as the destination ("`-`"), the junk probability is not written to standard output at the end of the transcript.

2002 September 17

The `Base64` decoder could hang if one of the lines it was decoding contained white space. Fixed.

Added logic to detect and discard header items which begin with our own *Xfile* sentinel. This shouldn't happen in the normal course of things, but somebody may try to spoof a downstream filter by sending mail which contains a sentinel purporting to be a classification by of of its legitimacy. Deleting our own header items also allow us to process our own transcripts containing them and reproduce the same results as if they hadn't been added.

Cleaned up the horrific ⟨ Activate MIME decoder if required 140 ⟩ section which "jes' grew" in **mailFolder** :: *nextLine* as more and more complexities were cranked in to MIME part decoding, multiple byte character sets, parsing ASCII strings out of binary data streams, etc.

2002 September 18

Cleaned up documentation of command line options, clarifying that they are logically commands which must be specified in the order in which they are to be executed. In the process, I added an example of invoking `annoyance-filter` as a pre-processor for a mail sorting program such as `Procmail` to the "Quick and dirty user guide".

Added a new `annoyance-filter-run` shell script to execute the program in default filter mode with the executable and dictionary installed in the default "`$HOME/.annoyance-filter`" directory. Oh, you haven't hear about that. . . well, stay tuned. . . details in the next episode.

Incremental refinement of the `README` and `INSTALL` files, with many keystrokes to go before we put these documents to sleep.

Added `--verbose` tell-tales for the `--plot` and `--statistics` options.

Replaced the `annoyance-filter.1` manual page with a cop-out which directs the esteemed reader to the PDF program documentation. This thing is changing so rapidly that the last thing I need is to maintain four copies of the bloody command line option documentation. *Four?* Think about it: the program (`CWEB`), its embedded `--help` option text, a Web page (nonexistent at the moment, thank Bob), and a manual page. Keeping all four simultaneously in sync is something which could appeal only to an accountant. I'm a programmer, not an accountant—I drink their blood, but I don't do their work.

The code which discards header lines we've generated attempted to remove lines from the transcript even when no transcript was being generated, for example, when adding a message we'd previously processed to the `--mail` or `--junk` database. This caused a Λ pointer reference in ⟨ Check for lines with our sentinel already present in the header 133 ⟩—fixed.

Hours of patient, unremunerated toil cleaning up `Makefile.in` to bash things into a distributable form. I added an `install` target which installs the program in the default `$HOME/.annoyance-filter` directory, creating a customised `run` program (`annoyance-filter-run` in the build directory) which supplies the home directory which `sendmail` doesn't. Massive clean-up of `Makefile.in`, yielding a template which is far more generic for our next foray into software land.

2002 September 19

Further testing revealed that the segmentation fault in **dictionary** :: *purge* which I thought I fixed a week or so ago was still lurking to bite the unwary soul whose dictionary contained a large number of words eligible for purging. As far as I can determine, when you *erase* an item from a **set**, not only does the iterator argument to the *erase* become invalid, in certain cases (but not always), an iterator to the *previous* item—not erased, becomes invalid, leading to perdition when you attempt to pick up the scan for purgable words from that point. After a second tussle with *remove_if*, no more fruitful than the last (for further detail, see the **dictionary** :: *purge* implementation, I gave up and rewrote *purge* to resume the scan from the *start* of the **set** every time it erases a member. This may not be efficient, but at least it doesn't crash! In circumstances where a large percentage of the dictionary is going to be purged, it would probably be better to scan for contiguous groups of words eligible for purging, then *erase* them with the flavour of the method which takes a start and end iterator, but given how infrequently `--purge` is likely to be used, I don't think it's worth the complication.

In a fit of false economy, I accidentally left the door open to the possibility that with an improbable albeit conceivable sequence of options we might try to classify a message without updating the the probabilities in the dictionary to account for words added in this run. I added calls on *updateProbability*( ) in the appropriate places to guarantee this cannot happen. The only circumstances in which this will result in redundant computation of probabilities is while building dictionaries, and the probability computation time is trivial next to the I/O and parsing in that process.

In the normal course of events the vast majority of runs of the program will load a single dictionary and use it to classify a single message. Since we've guaranteed that the probabilities will always be updated before they're written to a file, there's no need to recompute the probabilities when we're only importing a single dictionary. I added a check for this and optimised out the probability computation. When merging dictionaries with multiple `--read` and/or `--csvread` commands, the probability is recomputed after adding words to the dictionary.

If you used a dictionary in which rare words had not been removed with `--purge` to classify a message, you got screwball results because the -1 probability used to flag rare words was treated as if it were genuine. It occurred to me that folks building a dictionary by progressive additions might want to keep unusual words around on the possibility they'd eventually be seen enough times to assign a significant probability. I fixed ⟨ Classify message tokens by probability of significance 173 ⟩ to treat words with a probability of -1 as if they had not been found, this simulating the effect of a `--purge`. Minor changes were also required to CSV import to avoid confusion between rare words and the pseudo-word used to store message counts. Note that it's still more efficient to `--purge` the dictionary you use on classification runs, but if you don't want to keep separate purged and unpurged dictionaries around, you don't need to any more.

Added a new `--annotate` option, which takes an argument consisting of one or more single character flags (case insensitive) which request annotations to be added to the `--transcript`. The first such flag is "`w`", which adds the list of words and probabilities used to rank the message in the same form as included in the `--pdiag` report. To avoid duplication, I broke the code which generates the word list out into a new *addSignificantWordDiagnostics* method of **classifyMessage**.

Added a "`p`" annotation which causes parser diagnostics to be included in the `--transcript`. This gets rid of all the conditional compilation based on `PARSE_DEBUG` and automatically copies the diagnostics to standard error if *verbose* is set. Parser diagnostics are reported with the *reportParserDiagnostic* method of **mailFolder**; other classes which report errors do so via a pointer to the **mailFolder** they're acting on behalf of.

Well, my sleazy reset to the beginning trick for **dictionary** *purge* really was intolerably slow for real world dictionaries. I pitched the whole mess and replaced it with code which makes a **queue** of the words we wish to leave in the dictionary, then does a *clear* on the dictionary and re-*insert*s the items which survived. This is simple enough to entirely avoid **map** iterator hooliganism and runs like lightning, albeit using more memory.

Break out the champagne! The detestable `MIME_DEBUG` conditional compilation is now a thing of the past, supplanted by a new "`d`" `--annotate` flag. No need to recompile every time you're inclined to psychoanalyse a message the parser spit up.

Added a *name* method to **MIMEdecoder** and all its children, then took advantage of that to dispense with the horrific duplication of decoder diagnostic code in ⟨ Verify Content-Transfer-Encoding and activate decoder if necessary 147 ⟩. What was previously dispersed among the several branches of the decoder activation is now collected together in a single case after the decoder has been chosen.

Modified `Makefile.in` to delete the fussy `core.`*process* files Linux has taken to produce.

Fixed `configure.in` to specify `-Wall` if we're building with GCC.

2002 September 20

On Solaris, GCC is prone to hang if invoked with `-O2` (at least as of version 2.95.3). I twiddled the `configure.in` to change the compile option to `-O` for Solaris builds.

`ctangle` and `cweave` spewed copious warnings on a GCC `-Wall` build. To avoid modifying these programs, which are prefectly compliant ANSI C, I changed `Makefile.in` to suppress the `-Wall` option for them when the compiler is detected as GCC.

`make dist` didn't do a `make distclean` before generating the distribution archive, which could result in build-specific files being included in the archive. Fixed.

2002 September 21

Added documentation on how to integrate `annoyance-filter` into a `.forward` pipeline to `Procmail`, and build a `.procmailrc` rule set for a typical user-level filtering. It's 03:40 and I'm going to get some sleep before proofing this text—at the moment it's something between a random scribble and a first draft.

Okay, I just couldn't *stand it*...I just *had* to take another crack at the infernal **dictionary**::*purge* method. One of the many bees in my bonnet buzzed the idea into my ear that I could avoid both the extra memory consumption of yesterday's scheme and the risk of instability in the container by testing the probability of the first item in the **map**, adding it to the **queue** of survivors if its probability is significant, then performing an *erase*(*begin*( )). Cool, huh? No iterators, no mess, no two copies of any word in memory.

The hits just keep on coming...the stupid built-in purge in **dictionary**::*resetCat* also ran afoul of the "stale iterator" problem. I blew it away—henceforth, it's up to you to do a `--purge` after a `--clearmail` or `--clearjunk`. With the new tolerance for un-purged dictionaries, no great harm will be done if you forget.

Added a `\subsection` macro to create subheads within documentation sections. The section number is automatically grabbed from the `cwebmac.tex` definition, but lower level numbering is manual, permitting you to add additional levels of hierarchy with a specification like:
`\subsection{4.2.1}{Twiddling little details}`.

It turns out that all the cheesy mess I put in to patch the user's home directory into the `annoyance-filter-run` script wasn't necessary after all since `sendmail` is kind enough to change to the user's home directory before piping a message to a program. This means we can just `cd` to `.annoyance-filter` relative to the home directory. This also means one can remove the absolute path name from the `.forward` file, which cleans up the documentation on integration with `Procmail`.

Added a rather tacky `check` target to the `Makefile.in` to serve as a "sanity check" that doesn't require an extensive training databases. The scheme is to train the program with the source code for `annoyance-filter.w` serving as the mail collection and `statlib.w` the junk bucket. Then those programs themselves are tested, and the transcripts verified to confirm they were correctly classified. Astute observers will ask where I get off using something which isn't a well-formed mail folder to train the program. Well, it works thanks to a gimmick I put into the probability calculation to keep it from dividing by zero if one or both of the message counts were zero. That keeps anything untoward from happening when we're missing message headers, and the difference in the word content of the two files is so extreme that they reliably score correctly.

Added a new Perl gizmo, `TestFolder/testfolder.pl`, which walks through a mail folder, breaks out each message, and passes it through `annoyance-filter` to obtain the probability and classification. (The `annoyance-filter` command is defined by a string within the Perl program, so you can modify as you wish to evaluate the effects of other settings.) At the end of the folder, the total message count, number of messages scored as junk and mail, and the mean probability of messages in the folder are printed.

Added a "back" command to `SplitMail/splitmail.pl`. As you walk through a mail folder, the start address of each message you've seen is kept in a stack. The "`b`" command pops the stack and backs up to the previous message. This should reduce the pain when your sorting a folder and accidentally hit "`d`" when you meant to save the message somewhere. You can even go back after a search operation.

Moved the `splitmail.pl` and `testfolder.pl` from their own dedicated directories into a new `utilities` directory which `Makefile.in` includes in the archive. If and when these utilities require common code, such as the CSV parser, it will be easier to manage them all in the same directory.

Added help, requested by the "`?`" key, to `splitmail.pl` at both the disposition and the "more" prompt while viewing message text. If you assign additional folder destinations to disposition keys, they are automatically included in the help output.

Now that `splitmail.pl` is equipped with a "back" mechanism, there's no reason not to interpret a void disposition as a request to advance to the next message—if it's a fat-finger, just go back. Trolling through a target-sparse folder can now be done at the expense of only one keystroke per message.

2002 September 22

Went ahead and added code to dereference symbolic links up to 50 deep when deciding whether files are `gzip` compressed in **mailFolder**. What the heck, it's the solstice (well, it was a couple of hours ago) and the full Moon to boot—better to write silly code than trying to balance eggs on their little ends!

Much work on the documentation today, but little on the code. Slowly the python peristalsis moves us toward release.

2002 September 23

We're off to see the lizard, the wonderful lizard of WIN32! Naturally, all of our carefully crafted code to set up pipelines to decompress dictionaries evaporated under the harsh sun of WIN32. I added conditional compilation to disable everything that incompetent empire self-defined by its own *limes* and rusty Gates doesn't comprehend.

Building for WIN32 with DJGPP resulted in a natter about comparison of the *size_type* of a **multimap** to an **unsigned int**. The Linux compiler accepted this without a quibble. I added a **static_cast** to clear up the confusion.

OK, it built on WIN32 with DJGPP 2.953 and even passed the rudimentary tests I threw at it. So, I copied the executable back to the development directory, then discovered and fixed numerous bugs in the archive creation code in `Makefile.in` when the WIN32 distribution is enabled. Got better. A Zipped WIN32 build is now posted in the Web directory and linked to from the home page.

The `configure.in` script didn't check for the `-lm` math library. This somehow managed to work on Linux and Solaris, but failed on FreeBSD. I added the necessary `AC_CHECK_LIB` macro. (Reported by Neil Darlow).

Fixed several typos in the documentation of *computeJunkProbability* and reformatted the formula as a stacked fraction so it fits better on the page.

Added logic to `configure.in` to test for the presence of the `system` function and the `gnuplot` and `ppmtogif` utilities required by the `--plot` option. If any of them is missing, the option will be disabled when the program is compiled.

Added a test to `configure.in` for the presence of `readlink` and disabled the code that chases symbolic links in file name arguments if it's absent. I also added a "probable loop" warning if this code exceeds the maximum link depth limit.

Added a configurator test for the presence of `popen` and code to disable the ability to read compressed files if it's not present. This allowed me to remove the special case for `WIN32` I added last night to build on `DJGPP`—it's now subsumed into the test for `popen`.

Designed this version as "Release Candidate 1" and indicated this by setting `VERSION` to `"0.1-RC1"`.

Proofed the program documentation and the formatting of the code listing and fixed numerous typos and infelicitous layout.

Defined `-t` as a shortcut single-letter option for `--test` and `-r` as a shortcut for `--read`.

Release 0.1-RC1.

2002 September 24

Hugh Daniel took a look at the program and had many comments and suggestions. Until otherwise noted, the following items result from them.

Corrected "vertical interlace" terminology in the document to "vertical retrace". I'm forever screwing that one up.

Renamed `--purge` to `--prune`, which is a more precise (and less intimidating) description of what it does. For the moment, `--purge` is still accepted to ease the transition. Fixed the `check` target in `Makefile.in` to use `--prune`.

Added the hideous logic to `Makefile.in` to report overall pass/fail status for the `check` target.

Clarified the infectuous nature of the GPL in `COPYING`. While I was at it, I added information about the public domain status of DCDFlib.

Okay, back to self-generated items.... Changed the `--plot` option to use `pnmtopng` to generate the plot in PNG format instead of GIF.

Release 0.1-RC2.

2002 September 26

Added the ability to treat a directory as a mail folder consisting of messages in individual files in the directory. The contents of the directory are simply logically concatenated and are not restricted to one message per file–they may be UNIX mail folders in their own right.

After a huge amount of wasted effort trying to do this in an ultra-clean C++ fashion by defining an *idirstream* flavour of **istream** which returns the concatenated contents of files in a directory (I got *that close*, but couldn't make it work with the *getline* function for **string** without stooping to ugliness and making assumptions about the guts of the **iostream** package I believed unwarranted. This dead end is why you see no log entries for yesterday.

So, I ripped all that out and simply added logic to **mailFolder** to detect when it's passed a directory and wrap a loop traversing the directory around the main input loop; when end of file is encountered and we're traversing a directory, we look for the next file and commence processing it, declaring a genuine end of file only at the end of the directory.

This interacts in an interesting way with the MIME decoders. Recall that they are passed the actual **istream** from which the **mailFolder** normally reads and take charge of it until the end of the encoded section is reached. I added *no* logic to them specific to directory traversal—when they hit the end of the stream, they declare a missing terminator at the end of the section and bail out. But that's *good*—we don't want a missing terminator to gobble up the contents of a subsequent file in the directory folder. (Although if each file begins with a "`From␣`" line, it will cause the detector to bail out. This way, it's only after arriving back from the decoder that we detect we're at the end of a file in the directory and progress to the next item, if any, in the directory.

Yes, all of this is conditional on the presence of *opendir* and *stat*, which are required to detect and traverse the directory; the whole mess goes away if `configure.in` doesn't detect them. Yes, files in the directory may be compressed. And, yes, files in the directory may be symbolic links to compressed. But no, you can't recursively traverse directories; directories within a directory folder are simply ignored, which nicely avoids a special case for "`.`" and "`..`".

In the process of putting in all this junk, I discovered that the existing code for decompressing mail folders failed to call *pclose* to close out the pipeline, which is unkind. I added a destructor which makes sure it's called when necessary.

Added a new `fragmail.pl` program to the `utilities` directory. It splits up a monolithic mail folder into a directory with one message per file, making up file names from the message sequence in the input folder.

Added a new `signatures` target to `Makefile.in` which creates GnuPG signatures for each of the downloadable files and added a command to the `publish` target which copies them to the distribution directory.

Added code to `configure.in` to test for the presence of `pdftotext`, which we will eventually use to crack PDF files. Let's be realistic, however. This is cool (and will open the door to a general application specific binary file cracker, which I've been itching to do), but in terms to the mission statement of `annoyance-filter` and present day junk mail, is far from important. I've found precisely one PDF file in each of my mail and junk archives, so with a plane to catch tomorrow, I'm not going to stay up any later tonight worrying about refinements of this kind.

Release 0.1-RC3.

2002 September 29

Added logic to `Makefile.in` to prepare an HTML version of `man` page automatically from the `annoyance-filter.1` `troff` file. The output will require fixup since it is intended to be run from a CGI script, but should eliminate much of the duplication of labour inherent in maintaining parallel documentation in HTML and `man` page format.

2002 October 1

Expanded documentation of command line options in conjunction with preparation of a manual page using the `docutil/options.pl` translator.

Added "USAGE", "EXIT STATUS", and "FILES" sections to the manual page; all of these are specific to the man page and are not derived from `annoyance-filter.w`.

2002 October 2

Much work yesterday and today on automating the generation of documentation from the `CWEB` source file. I wrote a Perl program, `docutil/options.pl` to compile the options documentation from `annoyance-filter.w` into `troff` format with the `-man` macros. Actually, although containing special cases for the options, this is reasonably general and may be deployed for other common documentation in the future.

The output from `man2html` has some infelicitous links and formatting for HTML intended to be shipped with the product and included on its Web page. I wrote a Perl hack, `docutil/fixman2html.pl`, to correct these items, and modified the Makefile.in targets to generate a first draft HTML in annoyance-filter_man_raw.html, which is post-processed by the fixup program into the final annoyance-filter_ma file, which is now included in the distribution by the dist target and copied to the Web directory by publish, both of which targets generate it if necessary.

Added a mantroff target to Makefile.in to preview the troff format manual page using "groff␣-X" (if available on the system—if not, don't do that).

Wrote a docutil/cwebextract.pl Perl program which searches a CWEB file for a named section (which can be a regular "@" section, so long as the search target appears on the same line as the "@". If the section is found (matching is case insensitive and the search target given on the command line matches the first line containing a substring which it matches), the contents of the documentation section is written to standard output, trimming leading and trailing blank lines. The end of the documentation section is the next line which begins with an at sign or the end of file.

Moved the TEX definitions used to generate the options list to the top of annoyance-filter.w so they don't confuse the automatic extraction and translation process.

Modified docutil/cwebtex2man.pl to ignore TEX \bigskip commands, carefully avoiding generating a nugatory .PP in the troff output due to two consecutive blank lines once the command has been ignored.

Added the docutil directory and its contents to the distribution generation target in Makefile.in.

Generation of the "OPTIONS" section of the annoyance-filter.1 manual page from the corresponding section of annoyance-filter.w is now completely **Turbo Digital**TM. The invariant parts of the manual page are now defined in the "manual page macro" file annoyance-filter.manm. The Makefile.in now understands that annoyance-filter.1 is generated by processing this file with docutil/manm_expand.pl which expands \"%include statements in the macro file by extracting the specified section from the named CWEB file with docutil/cwebextract.pl, translating it into manual page troff with docutil/cwebtex2man.pl, and inserting it in the output file in place of the include statement. This completely eliminates all manual labour when updating the options in the manual page and guarantees that changes to the option documentation in annoyance-filter.w are propagated to the manual page document. The same mechanism can be used for other common documentation as the need arises.

2002 October 3

Subtly obfuscated the E-mail address to which bugs should be reported in the manual page so the process of transforming it into HTML won't result in a deadly mailto: link or a sniffable address in the page. Visual fidelity for human readers is maintained.

Updated the Web document to reflect the existence of the HTML manual page and added links to it.

Added a reference to the PDF document to the "SEE ALSO" section of annoyance-filter.manm. Fixed an embarrassing hyphenation of a file name by prefixing the offending word with the troff "don't hyphenate" escape "\%". (Apparently, even in nh mode, troff will hyphenate a word which contains an embedded hyphen unless you explicitly forbid it.)

Added the .w files to the winarch.zip archive used to transfer files to build for Win32. While they aren't strictly required, they're awfully handy to have should you encounter compile errors, which are reported with line numbers from the CWEB file. Looking it up while on Windows and patching the C++ file is a lot quicker than booting back into a real operating system to explore the problem.

In ⟨ Check whether folder is a directory of messages 124 ⟩ there was an erroneous reference to *dirFolder* not conditional on HAVE_DIRECTORY_TRAVERSAL—fixed.

The **mailFolder** constructor which accepts a file name in a **string** re-used the **ifstream** *isc*, which was previously used only when reading compressed files. This caused compile errors on systems where

`COMPRESSED_FILES` was not defined. We now unconditionally define *isc* in the **mailFolder** class definition.

With these fixes, the `makew32.bat` build on Win32 now works once again.

Added a `testw32.bat` file which runs a rudimentary test of the Win32 build similar to the `check` target in `Makefile.in`. I added this file to both the `dist` and `winarch` archive generation targets in `Makefile.in`.

Modified `Makefile.in` to replace the hard-coded `/ftp/annoyance-filter` destination with a `PUBDEST` declaration at the top of the file which defaults to the same directory. This permits overriding the default publication destination for use at another site or for nondestructive testing of new releases simply by editing the `Makefile`. Some day, it might make sense to permit overriding this with an option at `./configure` time, but this is not that day.

Release 0.1-RC4.

2002 October 11

Integrated the application string parsers for Flash and PDF formats, which were developed in a separate stand-alone test program. These include the classes **applicationStringParser** (mother of all application parsers), **flashStream**, **flashTextExtractor**, and **pdfTextExtractor**, the latter compiled in only if all the utilities it needs to decode PDF via a pipe to `pdftotext` are present. At the moment, these aren't hooked up to the mail folder, but are merely exercised by code in the `--jig`.

Integrated Knuth and Levy's `CWEB` version 3.64 in the `cweb` directory. The `CWEAVE` and `CTANGLE` programs are built with a change file, `common-bigger.ch` which increases the input line length limit to 400 characters as I did in the earlier 3.63 release.

Added plumbing to invoke Flash and PDF parsers for attachments with those application types. Thanks to the inability to take a class member function as an unqualified function pointer, this is somewhat tacky, requiring a pointer to the **mailFolder** to obtain decoded data.

2002 October 12

Added decoders and interpreters for Shift-JIS and Unicode (UCS-2, UTF-8, and UTF-16 encodings). These are used to decode and interpret these character sets in Flash animations whose fonts are so tagged.

Added logic to invoke the new Unicode UTF-8 decoder when a MIME part's `charset=` designates it so encoded.

2002 October 13

In the process of testing UTF-8 decoding of Unicode messages, I stumbled over a bug in ignoring HTML comments embedded within tokens, a common trick in junk mail to evade naïve filters, for example, "`remo<!---->ve␣your<!---->self`". (Yes, I know a valid HTML comment is supposed to contain a space after the initial and before the final sentinel, but junk mail often violates this rule, counting on sloppy browsers not to enforce the standard, so we must comply in the interest of "seeing what the user would".) HTML comments are now completely discarded, even when embedded within tokens.

The `dist` target in `Makefile.in` failed to clean the `cweb` directory before including it in the source archive, which could have the result of leaving objects and binaries not compatible with the system on which the user is installing. I modified the target to descend into the `cweb` directory and `make␣clean`. This promptly ran into another problem because the `CWEB Makefile` deletes the C source for `CWEAVE`, using the bootstrapped `CTANGLE` to re-build it. This is clean, but runs afoul of my rebuilding both programs directly in the outer `Makefile`. I saved the original `CWEB` makefile as `Makefile.ORIG` and modified the `clean` target in the actual `Makefile` to leave `cweave.c` around. I also modified our own `clean` target to clean the `cweb` directory as well.

Attempting to build `.dvi` or `pdf` targets after you'd cleaned the `cweb` directory failed for lack of `cweave`; I added a dependency to `Makefile.in` to ensure it's rebuilt when needed.

Since certain recent versions of `gcc` libraries have begun to natter if C++ include files specify the `.h` extension (which, for years, was *required* by those self-same libraries), I eliminated them from our list of includes, which finally seems to work on `gcc` 2.96. Doubtless this will torpedo somebody using an earlier version.

Broke up the unreadably monolithic list of include files into sections which explain what's what.

Dooooh! Forgot to disable the declaration of the **pdfTextExtractor** in **mailFolder** when `HAVE_PDF_DECODER` was not defined, which was the undoing of the Win32 build; fixed.

Release 0.1-RC5.

2002 October 19

Added a check in *classifyMessages* to verify that a dictionary has been loaded before attempting to classify a message. If no dictionary is present, a warning is written to standard error and the junk probability is returned as 0.5.

Added a warning if command line are specified after a `--classify` command. Since this command always exits with an exit code indicating the classification, specifying subsequent arguments is always an error.

Added a bunch of consistency checking for combinations of options which don't make any sense and suggest the user doesn't understand in which order they should be specified. To facilitate this, I modified the code for the `--classify` option to set a new *lastOption* flag to bail out of the option processing loop and set *exitStatus* to the classification rather than exiting directly before the option consistency checks are performed. This cleans up the control structure in any case.

In the process of adding the above code, I discovered that the *any*( ) method of **bitset** seems to be broken in the `glibc` which accompanies `gcc` 2.96. I tested *count*( ) against zero and that seems to work OK.

Implemented phrase tokens. You can consider phrases of consecutive tokens as primitive tokens by specifying the minimum and maximum words composing a phrase with the `--phrasemin` and `phrasemax` options. These default to 1 and 1, which suppresses all phrase-related flailing around. If set otherwise, tokens are assembled into a queue and all phrases within the length bounds are emitted as tokens. How well this works is a research question we may now address with the requisite tool in hand.

2002 October 20

Added code to import a binary dictionary file with the `--read` option using memory-mapped I/O if `./configure` detects that facility and defines `HAVE_MMAP`. This isn't a big win on individual runs of the program, but if you're installing it on a high volume server, multiple read-only references to the dictionary file (be sure to make the file read-only, by the way) can simply bring the file into memory where it is re-used by multiple instances of the program. (Of course, if the system has an efficient file system cache, that may work just as well, but there's no harm in memory mapping in any case.) Thanks to the C++ theologians who deprecated the incredibly useful *strstream* facility, which is precisely what you need to efficiently access a block of memory mapped data as a stream, I included a copy of the definition of this facility in `mystrstream.h` so we don't have to depend on the C++ library providing it.

I was a little worried about writing phrases in CSV format without quoting the fields, but I did an experiment with Excel and discovered it doesn't quote such fields either—it only uses quotes if the cell contains a comma or a quote (in which case it forces the quote by doubling it). Since our token definition doesn't permit either a comma or a quote within a token, we're still safe.

2002 October 21

Added a `--phraselimit` option to discard phrases longer than the specified limit on the fly. This prevents dictionary bloat due to "phrases" generated by concatenation of gibberish from headers and strings decoded from binary attachments. These will usually be eliminated by a `--prune`, but that doesn't help if the swap file's already filled up with garbage phrases before reaching the end of the mail folder. The default `--phraselimit` is 0, which imposes no limit on the length of phrases.

2002 October 22

When the default *getNextEncodedLine* of a **MIMEdecoder** encountered the "`From␣`" line of the next message in a mail folder, it failed to store the line as the part boundary, which in turn caused **mailFolder** to mis-count the number of messages in a folder being parsed when training. I fixed this, and in the process re-wrote an archaic C string test used in ⟨ Check for start of new message in folder 128 ⟩ to use a proper C++ **string** comparison.

Corrected some ancient URLs in `README`, and added information on the SourceForge project there and in `annoyance-filter.manm`.

Release 0.1-RC6.

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

ANNOYANCE-FILTER