A Selective Dissemination Service for Users Within a Computer Net

by

Mark Anthony Pinone

Submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

at the

Massachusetts Institute of Technology

May 1983

© Mark Anthony Pinone 1983

Signature of Author

Department of Electrical Engineering and Computer Science

May 16, 1983

Certified by

Thesis Supervisor

Accepted by

Chairman, Departmental Committee on Theses

Archives

1
A Selective Dissemination Service for
Users Within a Computer Net

by

Mark Anthony Pinone

Submitted to the
Department of Electrical Engineering and Computer Science
on May 16, 1983 in partial fulfillment of the requirements
for the Degree of Bachelor of Science

Abstract

The design and implementation of a selective dissemination service for users within the MIT computer net is presented. The service searches each article of the New York Times News Service for keywords which are supplied by the user in a filter specification. Filters are sent to the service via electronic mail. As filters are matched, articles are sent to users via electronic mail. The service will periodically check its incoming mail and update its filter set, but as far as a user is concerned, it maintains a standing query for filters. The user may, at any time, send the service a filter. This is a benefit from using electronic mail as a form of communication.

Keywords: selective, dissemination, filter

Thesis Supervisor: David Gifford
Title: Assistant Professor of Computer Science and Engineering

The author hereby grants M.I.T. permission to reproduce and to distribute copies of this thesis document in whole or in part.
Acknowledgments

Many people contributed to the success of this project, either directly or indirectly. Steve Berlin and Ralph Hyre implemented routines which formatted the raw data from the news line into separate articles. In addition, they were both helpful in answering various questions dealing with anything form the system level to the program design level.

Robert Scheifler proved to be very helpful in answering questions concerning CLU and the CLU library. He pointed out a few useful procedures which saved me hours of work.

Franklin Klein, who is working on a similar project for the IBM personal computer, helped me throughout the project. He was particularly helpful in its early stages, giving me some basic ideas for the filter grammar and the tree structure.

Most of all, I would like to thank Prof. David Gifford. He advised me throughout this project, providing useful suggestions and guidance. His easy going attitude and the fact that he always had my goal of graduating on time in mind, made working with him a pleasure.
# Table of Contents

Chapter One: Introduction 6

Chapter Two: Overall Design Specifications 8

2.1 The Mail System 8
2.2 The New York Times News Service 8
2.3 The Filter 9

Chapter Three: Program Design 13

3.1 The Driver 13
3.2 The Tree Structure 15
3.3 The NYT Stream 18
3.4 The Hash Table 19
3.5 The Filter Stream 19
3.6 The Filter Compiler 20
3.7 The Node Class Structure 21
3.8 The Search Routine 21

Chapter Four: Conclusions 23

4.1 Overall 23
4.2 Future Enhancements 24
  4.2.1 More Sources of Data 24
  4.2.2 A Subject Field 24
  4.2.3 Better Error Handling 25
  4.2.4 % Option 26

References 27

Appendix A: Specifications 28
# Table of Figures

| Figure 2-1: A Typical Filter | 9 |
| Figure 3-1: Modular Dependency Diagram | 14 |
| Figure 3-2: A Tree | 15 |
| Figure 3-3: A Top Level Node | 15 |
| Figure 3-4: A Node | 16 |
| Figure 3-5: A Section | 16 |
| Figure 3-6: A Leaf | 17 |
Chapter One

Introduction

Services are needed which will enable users within a computer system to sift through the information which is available to them. Since computing power is becoming cheaper, computers are becoming more and more accessible to people. This increased accessibility has given people access to huge amounts of information. On line databases can provide information ranging from stock market reports to community interests messages. This information, if not used properly, can be a hindrance instead of a help. As John Naisbitt, author of Megatrends puts it -

Information technology brings order to the chaos of information pollution and therefore gives value to data that would otherwise be useless. If users - through information utilities - can locate the information they need, they will pay for it. The emphasis of the whole information society shifts then from supply to selection.\textsuperscript{Naisbitt 82}

When users are within a computer net, they may have access to even more information. It doesn’t make sense for each each user to run a process which has to search through all of this information; it would be much better to have a central process which would disseminate, or distribute the information. If the total computing power of the net is looked at, this would be far more efficient. A central process would make one pass at the information; rather than having each user make a pass.

This thesis describes a selective dissemination service for users within a computer net. An implementation on the VAX 11/750\textsuperscript{1} for the MIT computer net is

\textsuperscript{1}VAX is a trademark of the Digital Equipment Corporation
presented. Presently, the New York Times News Service is the only source of information that this project utilizes.

There are four chapters to this thesis. Chapter 2 presents the overall design specifications. Chapter 3 describes the dependency among program modules, and then each module design. Chapter 4 summarizes results and gives suggestions for future enhancements.
Chapter Two

Overall Design Specifications

2.1 The Mail System

Communication between the user and the service is through the mail system. A user who wishes to "subscribe" to the service mails a filter\(^2\) to the service's network mail address.\(^3\) To cancel his subscription to the service, the user would send a blank filter.

This communication system has a number of advantages. First of all, it is proven and reliable. Mailers, for most operating systems, have been around for quite a long time -- time to catch most of the bugs in the system. Secondly, the mail system, since it is not a real time communication system, has the effect of isolating the user and the service from failures of the net or of other computers. As far as the user is concerned, the service is providing a standing query for filters. The user may send filters at any time. Also, the service may send articles at anytime, regardless of whether the user is ready to see them.

2.2 The New York Times News Service

The New York Times News Service is the soul source of information for this project, at this time. This news service is similar to standard news lines, such as

\(^2\) to be described in section 2.3

\(^3\) in this case, pinone@MIT-CLS
OVERALL DESIGN SPECIFICATIONS

Associated Press and United Press. Current news and some special interest stories will be sent from 10 to 12 hours a day, running at 75 baud. Ralph Hyre and Steve Berlin implemented routines to take the data from the line, and put each article in a separate, formatted form.

Each formatted article has ten header fields (one line apiece) followed by the actual text of the article. The header fields are: message-id, type, priority, date, category, subject, section, title, author, and source. Currently, the message-id and the date fields are the only fields which are not available for subscribers to match key words in. The user can search through all other fields by making the proper specification within the filter.

2.3 The Filter

```
From PINONE@MIT-XX Sun May 15 18:09:20 1983
Date: 15 May 1983 1808-EDT
From: Mark A. Pinone <PINONE@MIT-XX>
Subject: Subscription Request
To: pinone@MIT-CLS

#sentence Reagan,Defense & ~budget
#paragraph computers,budget
#priority urgent
```

Figure 2-1: A Typical Filter

Users specify the articles they are interested in by building and mailing the service a filter (see figure 2-1). A filter is an ascii file which contains key words to be matched in the article, as well as special parameters which enable the user to specify when the filter will be matched. It must satisfy the following grammar:
<filter> ::= <line> | <line> "\n" <filter>

<line> ::= <group> | <group> "&" <line>

<group> ::= <series> | <connector> " " <series> | "# priority " <word> | "# filter " <filename>

<connector> ::= "+type" | "# category" | "# subject" | "# section" | "# title" | "# author" | "# source" | "# sentence" | "# paragraph"

<series> ::= <wordlist> | <wordlist> "," <series>

<wordlist> ::= <word> | "!" <word> | "~" <word>

<word> ::= <alphanumeric> | <alphanumeric> "." | <alphanumeric> <word>

<alphanumeric> ::= "A" - "Z" | "a" - "z" | "0" - "9"

<filename> ::= a valid system filename

If any of the filter's lines are satisfied, the filter is satisfied. Lines, therefore, can be thought of as being logically ORed together. Every group within a line must be satisfied for the line to be satisfied. Groups are logically ANDed together.

Groups can have optional parameters in front of them. A parameter must have a "#" character as its first character, to distinguish it from any other word. The following parameters are recognized:

- #type - Match if the words are within the "type" header of the article.
- #category - Match if the words are within the "category" header of the article.
OVERALL DESIGN SPECIFICATIONS

- #subject - Match if the words are within the "subject" header of the article.

- #section - Match if the words are within the "section" header of the article.

- #title - Match if the words are within the "title" header of the article.

- #author - Match if the words are within the "author" header of the article.

- #sentence - Match if the words are within the same sentence.

- #paragraph - Match if the words are within the same paragraph.

- #priority - Match if the word is within the "priority" header of the article.

- #filter - Reads the next word as a file name for another filter and compiles the filter within the current one. If the file does not exist on the system where the service resides, a bad format error will occur.

Groups contain words which are to be matched within the article. Each word is either separated by a comma, or a space. Words which are separated by a comma may appear anywhere in the article, or header if the appropriate group parameter was specified. Words which are separated by a space must be adjacent to each other to be matched.

A word may also have a "~" or "!" in front of it, which negates the word. If the first word of a series of adjacent words are negated, then that whole series is considered negated. For example, "~community information" means that we do not want to see "community information"; it does not mean that we are looking for "information" without "community" preceding it.

If the negate occurs within a series of adjacent words, then just that word is negated.
So, "community ~information" means that we are looking for "community" without "information" following it. Also note that "community !information services" means that we are looking for a three word series, with "community" as the first word, "services" as the third, and any word except "information" as the second. The series "community services" would not satisfy the above specifications.
Chapter Three

Program Design

The main factor in making design decisions was speed. The service is projected to be used by 50 - 500 users. If we assume 10 keywords per user, the search space would be 500 - 5000 keywords. The system already has a couple of other processes running continuously on it, so the program must be efficient.

The program must also be flexible. The program should be able to be easily modified to run on another machine, perhaps using a different mailing system. Also, new sources of information should be able to be added to the service without too much difficulty.

3.1 The Driver

The program has seven basic modules (see figure 3-1) - the driver, the compiler, the search routine, the filter stream, the hash table, the node class structure, and the tree structure.

The driver is the top level routine. By making some system calls, it finds out if any new filters have been mailed and will save them. A filter is always saved in a special filter directory with the first 14 characters of its return address as the file name. Next a list of all the current filter files will be obtained. The filters are then compiled together to yield one hash table to search for matches, and a list of the filters which were bad. A standard bad format file is mailed to each user who provided a bad filter. Their filters are deleted from the filter directory, as well as users who
provided an empty filter.

Before we can start searching, we have to find the next article to look at. This will usually be the most recent article, although it is possible to get a bit behind if we just finished looking at a long article. It is also possible that there are no more new articles, in which case we just wait for a while a check again.

When the article is found and opened for reading, the search begins. The article is read by the search routine once, and all return address of the matching filters are returned. The article is then mailed out to those users, by making system calls and a call to the mail program. The program then goes back to the beginning of a loop to
see if there is any new mail.

### 3.2 The Tree Structure

![A Tree](image)

Figure 3-2: A Tree

![A Top Level Node](image)

Figure 3-3: A Top Level Node

A tree is made up of four basic parts (see figure 3-2) - top level nodes, nodes, sections, and leaves. Top level nodes (see figure 3-3) have a return `address` field (
where to mail the article if the filter is matched), a class field, which is always equal to "other" (only group nodes have special classes, more on that latter), and a sections field which contains a list of the node’s sections.

---

<table>
<thead>
<tr>
<th>parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
</tr>
<tr>
<td>sections</td>
</tr>
</tbody>
</table>

Figure 3-4: A Node

Nodes (see figure 3-4) are just like top level nodes except that they contain a parent field instead of a return address field. The parent field points to a section. A class field contains "sent" if the sentence parameter was given in the group specification, "parg" if the paragraph parameter was given, or "other" for all other node specifications, except a top node of a sub filter. A sub node filter is a filter which was specified by the "filter" parameter. The top node for a sub filter must be treated differently than a regular node. For it to be satisfied, only one of its sections have to be true.

---

<table>
<thead>
<tr>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
</tr>
<tr>
<td>negate</td>
</tr>
</tbody>
</table>

Figure 3-5: A Section

Sections (see figure 3-5) contain a node field which points to the node the section is contained in. A value field is either true or false, depending on whether the section
has been satisfied. The negate field will be true if the first character of the corresponding word series was a '~' or '!'.

Leaves (see figure 3-6) contain a parents field which contain pointers to sections. The text field contains the actual words to be matched. The category field contains a header category (type, category, subject, section, title, author, or source), if such a header parameter was within the group, otherwise it contains "text".

The tree structure is designed to be a multi-level structure with top nodes at the top level and leaves at the bottom. Any section's value may be set. If the section is a negate section, then the actual value stored will be the opposite of the one given. There are also routines to bubble and re-initialize. If a node's parent section is false when the node is being bubbled, it is set to true and the parent node is bubbled; otherwise the bubble call has no effect. When a top node is bubbled, all of its sections are set to true and the matched return address string is returned. By setting all of its sections to true, we are preventing a filter to be matched twice for the same article. A node whose class is "sub top" is bubbled the same way a top node is, except that instead of a return address being returned, its parent is bubbled (if needed).

Nodes and top nodes are re-initialized by setting all of their section values to false. Note that in the case of a negated section of a group node, the actual value set
would be true. Thus if a negated word is found, it will keep the node form bubbling; otherwise it won’t prevent a bubble.

3.3 The NYT Stream

The nyt stream is a data structure which is concerned with all the details of the article’s format. There are routines available to read the next word from the article, to peek at the next word in the article, to get the current category (header section or "text") we are in, and to re-initialize the peek.

The structure must have the ability to look ahead at the words on the actual stream, without actually reading them. This is accomplished through and internal buffer. After a peek, the word is put at the end of this buffer. A read will always take a word form the beginning of this buffer before it takes words from the actual stream. An index is also kept to show where in the buffer the next peek will take its word from. If this index is outside of the buffer, then the next peek word will be taken form the actual stream. Successive calls to peek will always return successive words. When re-initialize is called, the buffer index is set to the beginning of the buffer.

The structure keeps track of the current category. If we are in a header category, and get to the end of the line, the category will be changed to the next header and the next word will be the first word of that header (past the header title).

Sentence and paragraph boundaries are also recognized. When and end of sentence or paragraph is detected, special markers are placed on the buffer. A peek or read can latter pick up these special markers and signal their caller that a sentence or paragraph bound has occurred.
3.4 The Hash Table

The leaves of the tree are put in hash table. The hash table implemented uses separate chaining with unordered lists to handle collisions. Even though separate chaining is not space efficient as other methods (such as linear probing or random probing), it is faster. With a probing method, we have to contend with clustering problems, which slow down operation considerably, especially when the table starts to fill up. Separate chaining with ordered lists may be slightly faster, but the extra overhead involved in building the hash table would diminish that advantage.\textsuperscript{73}

Only the first word of a word series in a leaf's text field is hashed. So, if the leaf's text was "community information services", it would be hashed under "community". This is done since we are looking at words form the nyt stream one word at a time. The \textit{find} routine in the hash table structure will read only one word, and peek at more words if there are more words in the leaf's text. This way, if the last word in a series does not match, the peek can be re-initialized so the next leaf under that hash table entry can be examined for a match.

3.5 The Filter Stream

The filter stream is a data structure which is concerned with the details of the filter format. There are routines available to \textit{peek} and to \textit{read} the next character, to get the \textit{address}, to get the \textit{connector}, to get the next \textit{word}, to get the \textit{new filter}, to read the \textit{words} in a word series, and to read in extra \textit{blank lines}.

The \textit{peek} and \textit{read next character} routines will ignore spaces. The get address routine will return the return mailing address and read headers to the beginning of the actual filter. The get connector routine will return a parameter if the next
character in the filter is "#"; otherwise it will signal that there is not a connector. Only alphanumeric characters will be returned. Extra non-alphanumeric characters (other than "&" between groups and "#" preceding a connector) will cause a bad format error.

Word and words will return the next word and the words of a word series, respectively. Alphanumeric characters and periods (to deal with abbreviations) will be returned. Characters other than these within a word will cause a bad format error.

The get new filter routine will read the next word (includes all characters, except "&", ",", and newline) as a filename. The file will be opened, made into a filter data structure, and returned.

The user may use this option to select among some pre-defined system filters (presently, not defined). These filters could be about typical topics of current interest. For instance, a mid eastern filter may contain a line for each mid eastern country.

3.6 The Filter Compiler

The filter compiler is a recursive decent compiler. The compiler builds a tree structure whose leaves are inserted in the hash table. The tree structure is usually only four levels deep - top level nodes, line nodes, group nodes, and leaves. It may be deeper in some places if the user selected another filter in his filter specification. For each level of filter recursive definitions, the maximum depth of the tree is increased by two.

The compiler follows the grammar. There are routines to read a filter, a line, a group,
a series, a wordlist, and a priority. Most of the routines, except read filter, will expect to have some data left on the current filter line when called. If an end of file or an end of line is the first thing these routines read, a bad format error will result. The read filter routine command ignore blank lines (even within the filter definition) and signal end of file if no more data was on the filter stream.

3.7 The Node Class Structure

A node class is just a group of related nodes which can be built upon. Routines are available to add a node to a node class, to look at the elements within a class, to bubble all the nodes within a class, and to re-initialize all of the nodes in a class.

3.8 The Search Routine

The search routine takes an article and a hash table as input. It will go through each article and build up three node classes as leaves are matched - sentence, paragraph, and other. Each leaf matched may have several section parents. Each of these sections have a particular node they are in (a group node). If the node's class is a sentence, it is added to the sentence node class. A node with a paragraph is added to the paragraph node class, and all other nodes are added to the other node class.

The significance behind these classes they reflect whether the sentence or paragraph parameter was given in the filter's group specification. A group sentence node must be treated differently than any group node. A sentence node must be bubbled at every sentence bound, and then re-initialized. This must be done since we don't want matches in two different sentences to satisfy the node. Also, if one of the word series is negated in the group specification, the node can not be bubbled until the end of the sentence. This is the reason why a self bubbling tree structure would not
work.

Group paragraph nodes also have to be treated differently. At each paragraph bound, they must be bubbled and then re-initialized, for the same reasons sentence nodes have to.

This is a fairly efficient scheme, since only nodes which had at least one leaf match will be bubbled or re-initialized - we do not have to go through all of the group nodes after every sentence. The sentence and paragraph classes can be re-built after a bubble and re-initialization occur. We are not looking at every sentence node matched in the entire article, but only the sentence nodes matched in the current sentence.

At the end of the article, all node classes (sentence, paragraph, and other) are bubbled. All return return address of matched filters are then returned.
Chapter Four

Conclusions

4.1 Overall

Generally, I am pleased with the results of the project. I had time to implement everything I initially set out to do. My main regret is that am not going to be around long enough to see the service in full operation. There are a lot of variables (hash table size, waits between articles, how often to check incoming mail, etc.) which need to be "tweaked", when the system is running all out, to give optimal performance.

The data structures were designed for performance, and from what I saw of them, they performed well. With a few filters, each article took a second or two to process. The design proved to be very flexible. The only system dependent code deals with electronic mail, and is called by the driver. Also, there is only one module (the nyt stream) that would have to be modified if another source of information was added. The overall modular design made debugging relatively easy.

Using electronic mail as a form of communication turned out to be a definite plus. Once the service was running, I didn’t have to make any modifications to the code so users on different systems could use it. Instead, electronic mail provided the service with a pre-defined user abstraction - as far as the service is concerned, all users look alike.

Electronic mail also made the service more reliable. I was building upon a system which has proven its reliability over the years, rather than introducing another
source of error.

4.2 Future Enhancements

4.2.1 More Sources of Data

This is perhaps the most obvious and useful enhancement. The New York Times Service was chosen as the source of information for this project because it was readily available. The overall design is in no way limited to this source. Another very useful information source might be a community messages directory. The user could specify some special parameter before a group that would specify that the follow words are to searched for in that directory.

The main addition that would have to be made to the program is a module that reads these files and understands any special formats. The problem with written text is that there are no hard fast rules to find the end of a sentence or paragraph; it is often a matter of style. For instance, the NYT stream would not be able to find the end of a paragraph in this thesis; it is looking for a newline followed by four spaces.

4.2.2 A Subject Field

A subject field could be specified when mailing out articles to users. The problem with the system now is that a user would have no idea how his filter was matched. If the key words or the group specification was put in the subject field of the mail, the user could identify the information quickly.

This change is not to difficult to implement, but it may slow down the system considerably. Nodes would have to have backwards pointers (to the nodes or leaves under them) so that the key words could be found when a match occurred. Finding
the key words would introduce some extra overhead, but the big overhead would come when the article is mailed out. Currently, an alias list is built and an article is mailed to all the users who provided matching filters with one call to the mail program. Having different subject fields for the same article implies one call to the mail program for each filter matched.

Although this would be a useful enhancement, it could not be implemented until the system has been running for quite some time, to get an idea on how the computer resources were being used.

4.2.3 Better Error Handling

It would be useful for the user who sends a bad filter to get some type of message specifying why the filter was bad. Presently, a standard message is sent to all users who supply a bad filter.

This change, like the subject field change, would not be too difficult to implement. Instead of the compiler routines which detect bad format errors signaling just a bad format, they could return a string of explanation. This string, as well as the users filter, could be mailed to him.

The problem with implementing this change is that it will introduce more overhead. Instead of making one call to the mailer to mail out all the bad filter messages, we have to make a separate call for each bad filter. This is a similar problem that the subject field change would introduce, but it is not so severe. If we don't expect to get too much mail, the chances of getting more than one bad filter between save mail calls is slim. So with either implementation, there will most likely be only one call to the mail system at a time.

The only real overhead remaining would be the time needed to build the actual
message. A file would have to be written to include the old filter and the error message. This would not take too long, and since it could make the users life easier, would be well worth it.

4.2.4 % Option

A user who is interested in a certain sub-set of information may not want to see all of it. A "%" parameter could be specified at the end of a line that would be satisfied a percentage of the times the actual line node is satisfied. It would have the effect of a random selector within that sub-set of information.

This would not be very difficult to implement. A random number generator can be used to give the approximate desired results. If the routine is not satisfied, it will simply keep the line node from bubbling.
References

[Knuth 73]
Knuth, D. E.
Addison-Wesley, Reading, Ma., 1973.

[Naisbitt 82]
Naisbitt, John.
Megatrends: Ten New Directions Transforming Our Lives.
Appendix A

Specifications

driver = proc()
% requires: nil
% modifies: nil
% effect : Main driver of the program - will see if there are any
% new filters, compile if there are, then search the
% next article to see which filters matched. The
% matched filters will have the article mailed to their
% return address, and bad filters will have a bad filter
% message mailed to their return address, before being
% deleted.
end driver
APPENDIX A: SPECIFICATIONS

```plaintext
mail_out = proc(addrs:array[string], in_file:file_name)
  .signals(error(string))
  % requires: nil
  % modifies: nil
  % effect : Creates an alias list and
  %          mails the file to the addresses
  end mail_out

save_mail = proc() signals(error(string), no_new)
  % requires: File exit.mail exist
  % modifies: File out.mail, filter directory
  % effect : See if there is any new mail,
  %           if so save it in a file with
  %           the return address as a name.
  end save_mail
```
APPENDIX A: SPECIFICATIONS

wait = proc()
  % requires: nil
  % modifies: nil
  % effect : Does nothing for 15 seconds.
end wait

delete_filts = proc(kill_files:array[string]) signals(error(string))
  % requires: File in.rm exist (empty file)
  % modifies: The files which kill_files refer to
  % effect : Deletes the given filter files
end delete_filts

write_error = proc(error:string) signals(error(string))
  % requires: nil
  % modifies: The file error.txt
  % effect : Writes the error string to error.txt.
end write_error

get_filters = proc() returns(array[file_name])
  signals(error(string))
  % requires: File in.1 exists (empty file)
  % modifies: Files out.1 and error.1
  % effect : Gets the files in the filter directory
end get_filters

get_next_file = proc() returns(file_name) signals(error(string), none)
  % requires: file /usr/nyt/output/lastfile exist
  % modifies: nil
  % effect : Get the next article to read.
  % If there is none, signal none.
end get_next_file

next_article = proc(last:string) returns(string)
  % requires: nil
  % modifies: nil
  % effect : Given an article string, returns the next article string
end next_article

peek_char = proc(in_stream:stream) returns(char)
  signals(error(string), end_of_file)
  % requires: nil
  % modifies: in_stream
  % effect : Peeks at the next character (ignoring spaces).
end peek_char
**APPENDIX A: SPECIFICATIONS**

```plaintext
next_char = proc(in_stream:stream) returns(char)
    signals (error(string), end_of_file)
% requires: nil
% modifies: in_stream
% effect : Gets the next character (ignoring spaces).
end next_char

low_case = proc(in_string:string) returns(string)
% requires: nil
% modifies: nil
% effect : Returns the low case of a string.
end low_case
```
APPENDIX A: SPECIFICATIONS

search = proc(hash_table: hash_table_type,
               nyt_stream: nyt_stream_type,
               sent_nodes: node_class_type,
               parg_nodes: node_class_type)
    returns(array(string)) signals(error(string))
% requires: nil
% modifies: nyt_stream
% effect : Will search through the given nyt_stream and return a
% list of addresses which were matched
end search

add_matches = proc(mail_to:array[string], addrs:array[string])
% requires: nil
% modifies: mail_to
% effect : Adds addrs array (addresses which were just matched)
%          : to the mail_to array (all addresses matched so far).
end add_matches
nyt_stream_type = cluster is create, get_word, get.peek_word,
            re_init.peak, read.peek

% auxiliary information:
%
% The nyt stream knows about the details of articles form the New
% York Times News Service.
%
% representation = record[stm:stream,
%        buffer : array[string],
%        category: string,
%        peek_i : int]

% rep invariant: category one of type, priority, category, subject
% section, title, author, source, text.
% peek_i >= 1.
create = proc(in_stream:stream) returns(nyt_stream_type)
signals(error(string).end_of_file)
% requires: in_stream be open for read
% modifies: nil
% effect : Will create a new nyt_stream with in_stream as the
% stream, buffer initialized to an empty array,
% peek_i as 1. Will read past message id and point
% to the first word in the type category. Category
% is initialized to type.
end create

get_word = proc(nyt_stream:nyt_stream_type,
    sent_nodes:node_class_type,
    parg_nodes:node_class_type) returns(string)
signals(end_of_file, error(string))
% requires: nil
% modifies: nyt_stream
% effect : Reads the next word form nyt_stream. If the
% buffer is empty, the the word will come form
% nyt_stream.stm; otherwise, it will be the
% first word in the buffer.
end get_word

get.peek_word = proc(nyt_stream:nyt_stream_type) returns(string)
signals(error(string), end_of_file, bound)
% requires: nil
% modifies: nyt_stream
% effect : Will "peek" at the next word in nyt_stream. If
% peek_i > the high bound of the buffer, the next
% word will be read off of nyt_stream.stm, added
% to the buffer, returned, and peek_i will be
% incremented. If peek_i < the high bound of the
% buffer, buffer[peek_i] will be returned, and
% peek_i will be incremented.
APPENDIX A: SPECIFICATIONS

end get_peek_word

re_init_peek = proc(nyt_stream:nyt_stream_type)
  % requires: nil
  % modifies: nyt_stream
  % effect: Will set peek_i to 1.
end re_init_peek

get_category = proc(nyt_stream:nyt_stream_type) returns(string)
  % requires: nil
  % modifies: nil
  % effect: Get the current category
end get_category

find_next = proc(nyt_stream:nyt_stream_type) returns(string)
  signals(error(string), end_of_file)
  % requires: nil
  % modifies: nyt_stream
  % effect: Returns the next word in the stream, and will
  % change the category if needed.
end find_next

next_word = proc(nyt_stream:nyt_stream_type) returns(string)
  signals(error(string))
  % requires: nil
  % modifies: nyt_stream
  % effect: Returns the next word in the stream. If it came
  % across a sent or paragraph end, it will put a
  % special mark in the buffer.
end next_word

peek_alpha = proc(in_stm:stream) returns(char)
  signals(error(string), end_of_file)
  % requires: nil
  % modifies: in_stream
  % effect: Peeks at the next alphanumeric character.
end peek_alpha

peek_char = proc(in_stm:stream) returns(char)
  signals(error(string), end_of_file)
  % requires: nil
  % modifies: in_stm
  % effect: Peeks at the next alphanumeric character or
  % newline
end peek_char

end nyt_stream_type
APPENDIX A: SPECIFICATIONS

compile = proc(filters:array[filename])
    returns(hash_table_type,
            array[string])
    signals(error(string))
    % requires: filters be valid filter files which were mailed to the
    % service
    % modifies: nil
    % effect : Will compile all of the filters - build hash table and
    % and a list of bad format filters. Will delete any
    % empty filters from the filter directory.
end compile

read_filter = proc(filter : filter_type,
                    hash_table : hash_table_type,
                    top_node : node_type)
    signals(error(string), bad_format, end_of_file)
    % requires: nil
    % modifies: filter, hash_table, top_node
    % effect : Read in a filter. Uses the following grammar:
    %
    %  <filter> ::= <line> | <line>"\n"<filter>
end read_filter

read_line = proc(filter : filter_type,
                  hash_table : hash_table_type,
                  line_node : node_type)
    signals(error(string), bad_format)
    % requires: nil
    % modifies: filter, hash_table, line_node
    % effect : Reads in a line. Uses the following grammar:
    %
    %  <line> ::= <group>"&"<line>
end read_line

read_group = proc(filter : filter_type,
                   hash_table : hash_table_type,
                   line_node : node_type)
    signals(error(string), bad_format)
    % requires: nil
    % modifies: filter, hash_table, line_node
    % effect : reads in a line. Uses the following grammar:
    %
    %  <group> ::= <series> | <connector>" "<series> | 
    %    "#priority "<word> | "#filter "<filename>
end read_group

read_series = proc(filt:filter_type),

APPENDIX A: SPECIFICATIONS

```
hash_table:hash_table_type,
group_node:node_type,
category:string)
signals(error(string),bad_format)
% requires: nil
% modifies: filt, hash_table, group_node
% Read a series. Uses the following grammar:
% %
% <series> ::= <wordlist> | <wordlist>,"<series>
end read_series

read_wordlist = proc(filt:filter_type, hash_table:hash_table_type,
group_node: node_type, category:string)
signals (error(string), bad_format)
% requires: nil
% modifies: filt, hash_table, group_node
% effect : Read a wordlist. Uses the following grammar:
% %
% <wordlist> ::= <word> | "!"<word> | "~"<word>
% <word>" "<wordlist> |
% "!"<word"> "<wordlist> |
% "~"<word"> "<wordlist>
end read_wordlist

read_priority = proc(filt:filter_type,
hash_table:hash_table_type,
group_node:node_type)
signals (error(string), bad_format)
% Reads in a priority
end read_priority
```
filter_type = cluster is create, get_peek_char, get_addr,
            get_next_char, get_connector, get_new_filter,
            get_next_word, words, read_blank_line

% auxilliary information :
% The filter_type cluster knows the details of a filter format.
% Spaces are ignored. Case is not important.
% representation: stream

create = proc(filter_stream:stream) returns(filter_type)
    % requires: nil
    % modifies: nil
    % effect : Creates a filter_type
    end create

get_addr = proc(filter:filter_type) returns(string)
    signals (error(string), end_of_file)
    % requires: nil
    % modifies: filter
    % effect : Gets the return address of the sender
    end get_addr

read_blank_line = proc(filter:filter_type)
    signals (error(string), end_of_file)
    % requires: nil
    % modifies: filter
    % effect : Reads in blank lines and "------" lines
    % (which save mail will write in the file).
    end read_blank_line

get_connector = proc(filter:filter_type) returns(string)
    signals (error(string), none, bad_format)
    % requires: nil
    % modifies: filter
    % effect : Get a connector ( #<word> ).
    end get_connector

get_next_word = proc(filter:filter_type) returns(string)
    signals (error(string), bad_format)
    % requires: nil
    % modifies: filter
    % effect : Gets the next word. Uses following grammar:
    %
    %
    % <word> ::= <alphanumeric> | <alphanumeric>"." |
APPENDIX A: SPECIFICATIONS

% end get_next_word

get_new_filter = proc(filter:filter_type) returns(filter_type)
  signals (error(string), bad_format)
  % requires: nil
  % modifies: filter
  % effect : Gets the new filter file name, and creates
  %          and returns a new filter_type
end get_new_filter

words = iter(filter:filter_type) yields(string)
  signals(error(string), bad_format)
  % requires: nil
  % modifies: nil
  % effect : Yields the words within a wordlist.
  %          Signals bad_format if no words were found.
end words

get_next_char = proc(filter:filter_type) returns(char)
  signals(error(string), end_of_file)
  % requires: nil
  % modifies: filter
  % effect : Get the next character in the filter
  %          ( ignoring spaces )
end get_next_char

get.peek_char = proc(filter:filter_type) returns(char)
  signals(error(string), end_of_file)
  % requires: nil
  % modifies: nil
  % effect : Peek at the next character in the filter
  %          ( ignoring spaces )
end get.peek_char

end filter_type
APPENDIX A: SPECIFICATIONS

hash_table_type = cluster is create, insert, find_next_match

% % representation = array[array[leaf_type]] %

create = proc() returns(hash_table_type)
% requires: nil
% modifies: nil
% effect : Creates a new hash table with empty elements.
end create

insert = proc(hash_table:hashtable_type, words:array[string],
cat:string) returns(leaf_type)
% requires: nil
% modifies: hash_table
% effect : If the given words and category are already
% in hash_table, the leaf where they were found
% will be returned. Otherwise, a new leaf will
% be created, inserted into hash_table, and
% returned.
end insert

find_next_match = proc(hash_table:hashtable_type,
nyt_stream:nyt_stream_type,
sent_nodes: node_class_type,
parg_nodes: node_class_type,
other_nodes: node_class_type)
returns(leaf_type)
signals(error(string), sent_bound, parg_bound, end_of_file)
% requires: nil
% modifies: nyt_stream
% effect : Will read the next word(s) and return the leaf
% where a match was found in hash_table. If an
% end_of_file is encountered in nyt_stream before
% a word can be read, end_of_file is signaled. If
% a sentence or paragraph marker is read off the
% buffer, then sent_bound or parg_bound will be
% signaled.
end find

end hash_table_type
APPENDIX A: SPECIFICATIONS

node_class_type = cluster is create, add_node, re_init_nodes

% representation = array[nodetype]
%

create = proc() returns(node_class_type)
  % requires: nil
  % modifies: nil
  % effect: Will create an empty node_class
  end create
end node_class_type

add_node = proc(node_class:node_class_type, node:node_type)
  % requires: nil
  % modifies: node_class
  % effect: Adds the given node to node_class
  end add_node

re_init_nodes = proc(node_class:node_class_type)
  % requires: nil
  % modifies: node_class's nodes
  % effect: Will re_init every node of the node_class
  end re_init_nodes

bubble_nodes = proc(node_class:node_class_type)
  signals (matches(array[string]))
    % requires: nil
    % modifies: node_class's nodes
    % effect: Will bubble every node of node_class
    end bubble_nodes

end node_class_type
APPENDIX A: SPECIFICATIONS

node_type = cluster is create, create_top_level, get_parent, get_addr,
            add_section, bubble, re_init

% representation = record[ sections: array[section_type],
%                        class: string,
%                       top_level: top_level_type]
%
% top_level_type = variant[yes: string,
%                          no: section_type]
% representation invariant: class is one of sentence,
%                        paragraph, sub_top, or other
%
create = proc(parent:section_type, in_class: string)
    returns(nodetype)
    % requires: nil
    % modifies: nil
    % effect : Returns a new node with the given section as the
    % parent (no top_level), no internal sections, and
    % in_class as its class.
end create

create_top_level = proc(addr:string, in_class:string)
    returns(nodetype)
    % requires: nil
    % modifies: nil
    % effect : Returns a node with the given address
    % (yes toplevel), no internal sections, and
    % and class equal to in_class
end create_top_level

get_parent = proc(node:node_type) returns(section_type)
            signals(top_level)
    % requires: nil
    % modifies: nil
    % effect : Returns the node's parent - if the node is a
    % top_level node, top level is signaled
end get_parent

get_addr = proc(node:node_type) returns(string)
    % requires: node is a top level node
    % modifies: nil
    % effect : Will return the mailing address of the top level
    % node
end get_addr
APPENDIX A: SPECIFICATIONS

add_section = proc(node:node_type, section:section_type)
% requires: nil
% modifies: node
% effect: Adds a new section to the node's list of sections
end add_section

bubble = proc(node:node_type) signals(matched(string))
% requires: nil
% modifies: any or all of the node's ancestors
% effect: If all of the node's sections are true, the node's
% parents will be set to true. A recursive call to
% bubble will occur if the parent is set from false
% to true. Top level and sub top level nodes are
% bubbled if only one of their sections is true. A
% top level node will signal matched and return its
% address if it is bubbled
end bubble

re_init = proc(node:node_type)
% requires: nil
% modifies: node's sections
% effect: All of node's section's will be set to false
end re_init

% Gets the node's class

get_class = proc(node:node_type) returns(string)
return(node.class)
end get_class

end node_type
APPENDIX A: SPECIFICATIONS

section_type = cluster is create, set_value, get_value, get_node
%
%   representation = record[node :node_type,
%                         value :boolean,
%                         negate :boolean]
%
create = proc(node:node_type, neg:boolean) returns(section_type)
  % requires: nil
  % modifies: nil
  % effect : Creates a new section with node as the associated
            % node, negate = neg, and value = false
  end create

set_value = proc(section:section_type, val:boolean)
  signals(matched(string))
  % requires: nil
  % modifies: section
  % effect : Sets the sections value to the one given. If value
            % is set to true, will do a bubble if necessary.
            % If negate is true, will set value to be ~val.
  end set_value

get_value = proc(section:section_type) returns(boolean)
  % requires: nil
  % modifies: nil
  % effect : Returns the section's value
  end get_value

get_node = proc(section:section_type) returns(node_type)
  % requires: nil
  % modifies: nil
  % effect : Returns the section's associated node
  end get_node

end section_type
APPENDIX A: SPECIFICATIONS

leaf_type = cluster is create, parents, text, add_parent, get_category

% representation = record[parents : array[section_type],
  text : array[string],
  category: string]

% rep invariant: category one of type, priority, category,
% subject, section, title, author, source,
% text.

create = proc(words:array[string], cat:string) returns(leaf_type).
% requires: nil
% modifies: nil
% effect : Creates a new leaf, with the given text and
category, and no parents.
end create

parents = iter(leaf:leaftype) yields(section_type)
% requires: nil
% modifies: nil
% effect : Will yield each of the leaf's parents.
end parents

add_parent = proc(leaf:leaf_type, parent:section_type)
% requires: nil
% modifies: leaf
% effect : Adds a parent to the leaf's list of parents.
end add_parent

text = iter(leaf:leaf_type) yields(string, bool)
% requires: nil
% modifies: nil
% effect : Will yield each word and whether it is negated
% which is part of the leaf's text.
end text

get_category = proc(leaf:leaf_type) returns(string)
% requires: nil
% modifies: nil
% effect : returns the leaf's category
end get_category

end leaf_type