Nuweb Version 1.56
A Simple Literate Programming Tool

Preston Briggs[1]
preston@tera.com
HTML scrap generator by John D. Ramsdell
ramsdell@mitre.org
Scrap formatting by Marc W. Mengel
mengel@fnal.gov
Continuing maintenance by Simon Wright
simon@pushface.org
and Keith Harwood
Keith.Harwood@vitalmis.com

[1]This work has been supported by ARPA, through ONR grant N00014-91-J-1989.

Contents

Chapter 1

Introduction

In 1984, Knuth introduced the idea of *literate programming* and described a pair of tools to support the practise [**?**]. His approach was to combine Pascal code with TEX documentation to produce a new language, `WEB`, that offered programmers a superior approach to programming. He wrote several programs in `WEB`, including `weave` and `tangle`, the programs used to support literate programming. The idea was that a programmer wrote one document, the web file, that combined documentation (written in TEX [**?**]) with code (written in Pascal).

Running `tangle` on the web file would produce a complete Pascal program, ready for compilation by an ordinary Pascal compiler. The primary function of `tangle` is to allow the programmer to present elements of the program in any desired order, regardless of the restrictions imposed by the programming language. Thus, the programmer is free to present his program in a top-down fashion, bottom-up fashion, or whatever seems best in terms of promoting understanding and maintenance.

Running `weave` on the web file would produce a TEX file, ready to be processed by TEX. The resulting document included a variety of automatically generated indices and cross-references that made it much easier to navigate the code. Additionally, all of the code sections were automatically prettyprinted, resulting in a quite impressive document.

Knuth also wrote the programs for TEX and METAFONT entirely in `WEB`, eventually publishing them in book form [**?**, **?**]. These are probably the largest programs ever published in a readable form.

Inspired by Knuth's example, many people have experimented with `WEB`. Some people have even built web-like tools for their own favorite combinations of programming language and typesetting language. For example, `CWEB`, Knuth's current system of choice, works with a combination of C (or C++) and TEX [**?**]. Another system, FunnelWeb, is independent of any programming language and only mildly dependent on TEX [**?**]. Inspired by the versatility of FunnelWeb and by the daunting size of its documentation, I decided to write my own, very simple, tool for literate programming.[1]

1.1 Nuweb

Nuweb works with any programming language and LATEX [**?**]. I wanted to use LATEX because it supports a multi-level sectioning scheme and has facilities for drawing figures. I wanted to be able to work with arbitrary programming languages because my friends and I write programs in many languages (and sometimes combinations of several languages), *e.g.,* C, Fortran, C++, yacc, lex, Scheme, assembly, Postscript, and so forth. The need to support arbitrary programming languages has many consequences:

No prettyprinting Both `WEB` and `CWEB` are able to prettyprint the code sections of their documents because they understand the language well enough to parse it. Since we want to use *any* language, we've got to abandon this feature. However, we do allow particular individual formulas or fragments of LATEX

[1] There is another system similar to mine, written by Norman Ramsey, called *noweb* [**?**]. It perhaps suffers from being overly Unix-dependent and requiring several programs to use. On the other hand, its command syntax is very nice. In any case, nuweb certainly owes its name and a number of features to his inspiration.

code to be formatted and still be parts of output files. Also, keywords in scraps can be surrounded by `@_` to have them be bold in the output.

No index of identifiers Because `WEB` knows about Pascal, it is able to construct an index of all the identifiers occurring in the code sections (filtering out keywords and the standard type identifiers). Unfortunately, this isn't as easy in our case. We don't know what an identifier looks like in each language and we certainly don't know all the keywords. (On the other hand, see the end of Section 1.2.2)

Of course, we've got to have some compensation for our losses or the whole idea would be a waste. Here are the advantages I can see:

Simplicity The majority of the commands in `WEB` are concerned with control of the automatic prettyprinting. Since we don't prettyprint, many commands are eliminated. A further set of commands is subsumed by LaTeX and may also be eliminated. As a result, our set of commands is reduced to only four members (explained in the next section). This simplicity is also reflected in the size of this tool, which is quite a bit smaller than the tools used with other approaches.

No prettyprinting Everyone disagrees about how their code should look, so automatic formatting annoys many people. One approach is to provide ways to control the formatting. Our approach is simpler—we perform no automatic formatting and therefore allow the programmer complete control of code layout. We do allow individual scraps to be presented in either verbatim, math, or paragraph mode in the TeX output.

Control We also offer the programmer complete control of the layout of his output files (the files generated during tangling). Of course, this is essential for languages that are sensitive to layout; but it is also important in many practical situations, *e.g.,* debugging.

Speed Since nuweb doesn't do too much, the nuweb tool runs quickly. I combine the functions of `tangle` and `weave` into a single program that performs both functions at once.

Page numbers Inspired by the example of noweb, nuweb refers to all scraps by page number to simplify navigation. If there are multiple scraps on a page (say, page 17), they are distinguished by lower-case letters (*e.g.,* 17a, 17b, and so forth).

Multiple file output The programmer may specify more than one output file in a single nuweb file. This is required when constructing programs in a combination of languages (say, Fortran and C). It's also an advantage when constructing very large programs that would require a lot of compile time.

This last point is very important. By allowing the creation of multiple output files, we avoid the need for monolithic programs. Thus we support the creation of very large programs by groups of people.

A further reduction in compilation time is achieved by first writing each output file to a temporary location, then comparing the temporary file with the old version of the file. If there is no difference, the temporary file can be deleted. If the files differ, the old version is deleted and the temporary file renamed. This approach works well in combination with `make` (or similar tools), since `make` will avoid recompiling untouched output files.

1.1.1 Nuweb and HTML

In addition to producing LaTeX source, nuweb can be used to generate HyperText Markup Language (HTML), the markup language used by the World Wide Web. HTML provides hypertext links. When an HTML document is viewed online, a user can navigate within the document by activating the links. The tools which generate HTML automatically produce hypertext links from a nuweb source.

(Note that hyperlinks can be included in LaTeX using the `hyperref` package. This is now the preferred way of doing this and the HTML processing is not up to date.)

1.2 Writing Nuweb

The bulk of a nuweb file will be ordinary LaTeX. In fact, any LaTeX file can serve as input to nuweb and will be simply copied through, unchanged, to the documentation file—unless a nuweb command is discovered. All nuweb commands begin with an "at-sign" (`@`). Therefore, a file without at-signs will be copied unchanged. Nuweb commands are used to specify *output files,* define *fragments,* and delimit *scraps.* These are the basic features of interest to the nuweb tool—all else is simply text to be copied to the documentation file.

1.2.1 The Major Commands

Files and fragments are defined with the following commands:

`@o` ***file-name flags scrap*** Output a file. The file name is terminated by whitespace.

`@d` ***fragment-name scrap*** Define a fragment. The fragment name is terminated by a return or the beginning of a scrap.

`@q` ***fragment-name scrap*** Define a fragment. The fragment name is terminated by a return or the beginning of a scrap. This a quoted fragment.

A specific file may be specified several times, with each definition being written out, one after the other, in the order they appear. The definitions of fragments may be similarly specified piecemeal.

A fragment name may have embedded parameters. The parameters are denoted by the sequence `@'value@'` where `value` is an uninterpreted string of characters (although the sequence `@@` denotes a single `@` character). When a fragment name is used inside a scrap the parameters may be replaced by an argument which may be a different literal string, a fragment use, an embedded fragment or by a parameter use.

The difference between a quoted fragment (`@q`) and an ordinary one (`@d`) is that inside a quoted fragment fragments are not expanded on output. Rather, they are formatted as uses of fragments so that the output file can itself be `nuweb` source. This allows you to create files containing fragments which can undergo further processing before the fragments are expanded, while also describing and documenting them in one place.

You can have both quoted and unquoted fragments with the same name. They are written out in order as usual, with those introduced by `@q` being quoted and those with `@d` expanded as normal.

In quoted fragments, the `@f` filename is quoted as well, so that when it is expanded it refers to the finally produced file, not any of the intermediate ones.

Scraps

Scraps have specific begin markers and end markers to allow precise control over the contents and layout. Note that any amount of whitespace (including carriage returns) may appear between a name and the beginning of a scrap. Scraps may also appear in the running text where they are formatted (almost) identically to their use in definitions, but don't appear in any code output.

`@{`*anything*`@}` where the scrap body includes every character in *anything*—all the blanks, all the tabs, all the carriage returns. This scrap will be typeset in verbatim mode. Using the `-l` option will cause the program to typeset the scrap with the help of LaTeX's `listings` package.

`@[`*anything*`@ ]` where the scrap body includes every character in *anything*—all the blanks, all the tabs, all the carriage returns. This scrap will be typeset in paragraph mode, allowing sections of TeX documents to be scraps, but still be prettyprinted in the document.

`@(`*anything*`@)` where the scrap body includes every character in *anything*—all the blanks, all the tabs, all the carriage returns. This scrap will be typeset in math mode. This allows this scrap to contain a formula which will be typeset nicely.

Inside a scrap, we may invoke a fragment.

`@<`*fragment-name*`@>` This is a fragment use. It causes the fragment *fragment-name* to be expanded inline as the code is written out to a file. It is an error to specify recursive fragment invocations.

`@<`*fragment-name*`@( `*a1*` @, `*a2*` @) @>` This is the old form of parameterising a fragment. It causes the fragment *fragment-name* to be expanded inline with the arguments *a1*, *a2*, etc. Up to 9 arguments may be given.

`@1, @2, ..., @9` In a fragment causes the n'th fragment argument to be substituted into the scrap. If the argument is not passed, a null string is substituted. Arguments can be passed in two ways, either embedded in the fragment name or as the old form given above.

An embedded argument may specified in four ways.

`@'string@'` The string will be copied literally into the called fragment. It will not be interpreted (except for `@@` converted to `@`).

`@<`*fragment-name*`@>` The fragment will be expanded in the usual fashion and the results passed to the called fragment.

`@{text@}` The text will be expanded as normal and the results passed to the called fragment. This behaves like an anonymous fragment which has the same arguments as the calling fragment. Its principle use is to combine text and arguments into one argument.

`@1, @2, ..., @9` The argument of the calling fragment will be passed to the called fragment and expanded in there.

If an argument is used but there is no corresponding parameter in the fragment name, the null string is substituted. But what happens if there is a parameter in the full name of a fragment, but a particular application of the fragment is abbreviated (using the `. . .` notation) and the argument is missed? In that case the argument is replaced by the string given in the definition of the fragment.

In the old form the parameter may contain any text and will be expanded as a normal scrap. The two forms of parameter passing don't play nicely together. If a scrap passes both embedded and old form arguments the old form arguments are ignored.

`@x`*label*`@x` Marks a place inside a scrap and associates it to the label (which can be any text not containing a @). Expands to the reference number of the scrap followed by a numeric value. Outside scraps it expands to the same value. It is used so that text outside scraps can refer to particular places within scraps.

`@f` Inside a scrap this is replaced by the name of the current output file.

`@t` Inside a scrap this is replaced by the title of the fragment as it is at the point it is used, with all parameters replaced by actual arguments.

`@#` At the beginning of a line in a scrap this will suppress the normal indentation for that line. Use this, for example, when you have a `#ifdef` inside a nested scrap. Writing `@##ifdef` will cause it to be lined up on the left rather than indented with the rest of its code.

Note that fragment names may be abbreviated, either during invocation or definition. For example, it would be very tedious to have to type, repeatedly, the fragment name

```
@d Check for terminating at-sequence and return name if found
```

Therefore, we provide a mechanism (stolen from Knuth) of indicating abbreviated names.

```
@d Check for terminating...
```

Basically, the programmer need only type enough characters to identify the fragment name uniquely, followed by three periods. An abbreviation may even occur before the full version; nuweb simply preserves the longest version of a fragment name. Note also that blanks and tabs are insignificant within a fragment name; each string of them is replaced by a single blank.

Sometimes, for instance during program testing, it is convenient to comment out a few lines of code. In C or Fortran placing `/* ... */` around the relevant code is not a robust solution, as the code itself may contain comments. Nuweb provides the command

`@%`

only to be used inside scraps. It behaves exactly the same as `%` in the normal LaTeX text body.

When scraps are written to a program file or a documentation file, tabs are expanded into spaces by default. Currently, I assume tab stops are set every eight characters. Furthermore, when a fragment is expanded in a scrap, the body of the fragment is indented to match the indentation of the fragment invocation. Therefore, care must be taken with languages (*e.g.,* Fortran) that are sensitive to indentation. These default behaviors may be changed for each output file (see below).

Flags

When defining an output file, the programmer has the option of using flags to control output of a particular file. The flags are intended to make life a little easier for programmers using certain languages. They introduce little language dependences; however, they do so only for a particular file. Thus it is still easy to mix languages within a single document. There are four "per-file" flags:

`-d` Forces the creation of `#line` directives in the output file. These are useful with C (and sometimes C++ and Fortran) on many Unix systems since they cause the compiler's error messages to refer to the web file rather than to the output file. Similarly, they allow source debugging in terms of the web file.

`-i` Suppresses the indentation of fragments. That is, when a fragment is expanded within a scrap, it will *not* be indented to match the indentation of the fragment invocation. This flag would seem most useful for Fortran programmers.

`-t` Suppresses expansion of tabs in the output file. This feature seems important when generating `make` files.

`-c`*x* Puts comments in generated code documenting the fragment that generated that code. The *x* selects the comment style for the language in use. So far the only valid values are `c` to get `C` comment style, `+` for `C++` and `p` for `Perl`. (`Perl` commenting can be used for several languages including `sh` and, mostly, `tcl`.) If the global `-x` cross-reference flag is set the comment includes the page reference for the first scrap that generated the code.

1.2.2 The Minor Commands

We have some very low-level utility commands that may appear anywhere in the web file.

`@@` Causes a single "at sign" to be copied into the output.

`@_` Causes the text between it and the next `@_` to be made bold (for keywords, etc.)

`@i` *file-name* Includes a file. Includes may be nested, though there is currently a limit of 10 levels. The file name should be complete (no extension will be appended) and should be terminated by a carriage return. Normally the current directory is searched for the file to be included, but this can be varied using the `-I` flag on the command line. Each such flag adds one directory tothe search path and they are searched in the order given.

`@r`*x* Changes the escape character '@' to '*x*'. This must appear before any scrap definitions.

`@v` Always replaced by the string established by the `-V` flag, or a default string if the flag isn't given. This is intended to mark versions of the generated files.

Finally, there are three commands used to create indices to the fragment names, file definitions, and user-specified identifiers.

`@f` Create an index of file names.

`@m` Create an index of fragment names.

`@u` Create an index of user-specified identifiers.

I usually put these in their own section in the LaTeX document; for example, see Chapter 5.

Identifiers must be explicitly specified for inclusion in the `@u` index. By convention, each identifier is marked at the point of its definition; all references to each identifier (inside scraps) will be discovered automatically. To "mark" an identifier for inclusion in the index, we must mention it at the end of a scrap. For example,

```
@d a scrap @{
Let's pretend we're declaring the variables FOO and BAR
inside this scrap.
@| FOO BAR @}
```

I've used alphabetic identifiers in this example, but any string of characters (not including whitespace or `@` characters) will do. Therefore, it's possible to add index entries for things like `<<=` if desired. An identifier may be declared in more than one scrap.

In the generated index, each identifier appears with a list of all the scraps using and defining it, where the defining scraps are distinguished by underlining. Note that the identifier doesn't actually have to appear in the defining scrap; it just has to be in the list of definitions at the end of a scrap.

1.3 Sectioning commands

For larger documents the indexes and usage lists get rather unwieldy and problems arise in naming things so that different things in different parts of the document don't get confused. We have a sectioning command which keeps the fragment names and user identifiers separate. Thus, you can give a fragment in one section the same name as a fragment in another and the two won't be confused or connected in any way. Nor will user identifiers defined in one section be referenced in another. Except for the fact that scraps in successive sections can go into the same output file, this is the same as if the sections came from separate input files.

However, occasionally you may really want fragments from one section to be used in another. More often, you will want to identify a user identifier in one section with the same identifier in another (as, for example, a header file defined in one section is included in code in another). Extra commands allow a fragment defined in one section to be accessible from all other sections. Similarly, you can have scraps which define user identifiers and export them so that they can be used in other sections.

`@s` Start a new section.

`@S` Close the current section and don't start another.

`@d+` **fragment-name scrap** Define a fragment which is accessible in all sections, a global fragment.

`@D+` Likewise

`@q+` Likewise

`@Q+` Likewise

`@m+` Create an index of all such fragments.

`@u+` Create an index of globally accessible user identifiers.

There are two kinds of section, the base section which is where normally everything goes, and local sections which are introduced with the `@s` command. A local section comprises everything from the command which starts it to the one which ends it. A `@s` command will start a new local section. A `@S` command closes the current local section, but doesn't open another, so what follows goes into the base section. Note that fragments defined in the base section aren't global; they are accessible only in the base section, but they are accessible regardless of any local sections between their definition and their use.

Within a scrap:

`@<+`*fragment-name*`@>` Expand the globally accessible fragment with that name, rather than any local fragment.

`@+` Like `@|` except that the identifiers defined are exported to the global realm and are not directly referenced in any scrap in any section (not even the one where they are defined).

`@-` Like `@|` except that the identifiers are imported to the local realm. The cross-references show where the global variables are defined and defines the same names as locally accesible. Uses of the names within the section will point to this scrap.

Note that the `+` signs above are part of the commands. They are not part of the fragment names. If you want a fragment whose name begins with a plus sign, leave a space between the command and the name.

1.4 Running Nuweb

Nuweb is invoked using the following command:

`nuweb` *flags file-name...*

One or more files may be processed at a time. If a file name has no extension, `.w` will be appended. LaTeX suitable for translation into HTML by LaTeX2HTML will be produced from files whose name ends with `.hw`, otherwise, ordinary LaTeX will be produced. While a file name may specify a file in another directory, the resulting documentation file will always be created in the current directory. For example,

```
nuweb /foo/bar/quux
```

will take as input the file `/foo/bar/quux.w` and will create the file `quux.tex` in the current directory.

By default, nuweb performs both tangling and weaving at the same time. Normally, this is not a bottleneck in the compilation process; however, it's possible to achieve slightly faster throughput by avoiding one or another of the default functions using command-line flags. There are currently three possible flags:

`-t` Suppress generation of the documentation file.

`-o` Suppress generation of the output files.

`-c` Avoid testing output files for change before updating them.

Thus, the command

```
nuweb -to /foo/bar/quux
```

would simply scan the input and produce no output at all.

There are several additional command-line flags:

`-v` For "verbose", causes nuweb to write information about its progress to `stderr`.

`-n` Forces scraps to be numbered sequentially from 1 (instead of using page numbers). This form is perhaps more desirable for small webs.

`-s` Doesn't print list of scraps making up each file following each scrap.

`-d` Print "dangling" identifiers – user identifiers which are never referenced, in indices, etc.

`-p` *path* Prepend *path* to the filenames for all the output files.

`-l` Use the `listings` package for formatting scraps. Use this if you want to have a pretty-printer for your scraps. In order to e.g. have pretty Perl scraps, include the following LaTeX commands in your document:

```
\usepackage{listings}
...
\lstset{extendedchars=true,keepspaces=true,language=perl}
```

See the `listings` documentation for a list of formatting options. Be sure to include a `\usepackage{listings}` in your document.

`-V` *string* Provide *string* as the replacement for the @v operation.

1.5 Generating HTML

Nikos Drakos' LaTeX2HTML Version 0.5.3 [**?**] can be used to translate LaTeX with embedded HTML scraps into HTML. Be sure to include the document-style option `html` so that LaTeX will understand the hypertext commands. When translating into HTML, do not allow a document to be split by specifying "`-split 0`". You need not generate navigation links, so also specify "`-no_navigation`".

While preparing a web, you may want to view the program's scraps without taking the time to run LaTeX2HTML. Simply rename the generated LaTeX source so that its file name ends with `.html`, and view that file. The documentations section will be jumbled, but the scraps will be clear.

(Note that the HTML generation is not currently maintained. If the only reason you want HTML is ti get hyperlinks, use the LaTeX `hyperref` package and produce your document as PDF via `pdflatex`.)

1.6 Restrictions

Because nuweb is intended to be a simple tool, I've established a few restrictions. Over time, some of these may be eliminated; others seem fundamental.

- The handling of errors is not completely ideal. In some cases, I simply warn of a problem and continue; in other cases I halt immediately. This behavior should be regularized.
- I warn about references to fragments that haven't been defined, but don't halt. The name of the fragment is included in the output file surrounded by <> signs. This seems most convenient for development, but may change in the future.
- File names and index entries should not contain any `@` signs.
- Fragment names may be (almost) any well-formed TeX string. It makes sense to change fonts or use math mode; however, care should be taken to ensure matching braces, brackets, and dollar signs. When producing HTML, fragments are displayed in a preformatted element (PRE), so fragments may contain one or more A, B, I, U, or P elements or data characters.
- Anything is allowed in the body of a scrap; however, very long scraps (horizontally or vertically) may not typeset well.
- Temporary files (created for comparison to the eventual output files) are placed in the current directory. Since they may be renamed to an output file name, all the output files should be on the same file system as the current directory. Alternatively, you can use the `-p` flag to specify where the files go.
- Because page numbers cannot be determined until the document has been typeset, we have to rerun nuweb after LaTeX to obtain a clean version of the document (very similar to the way we sometimes have to rerun LaTeX to obtain an up-to-date table of contents after significant edits). Nuweb will warn (in most cases) when this needs to be done; in the remaining cases, LaTeX will warn that labels may have changed.

Very long scraps may be allowed to break across a page if declared with `@O` or `@D` (instead of `@o` and `@d`). This doesn't work very well as a default, since far too many short scraps will be broken across pages; however, as a user-controlled option, it seems very useful. No distinction is made between the upper case and lower case forms of these commands when generating HTML.

1.7 Acknowledgements

Several people have contributed their times, ideas, and debugging skills. In particular, I'd like to acknowledge the contributions of Osman Buyukisik, Manuel Carriba, Adrian Clarke, Tim Harvey, Michael Lewis, Walter Ravenek, Rob Shillingsburg, Kayvan Sylvan, Dominique de Waleffe, and Scott Warren. Of course, most of these people would never have heard or nuweb (or many other tools) without the efforts of George Greenwade.

Since maintenance has been taken over by Marc Mengel, Simon Wright and Keith Harwood online contributions have been made by:

- Walter Brown `<wb@fnal.gov>`
- Nicky van Foreest `<n.d.vanforeest@math.utwente.nl>`
- Javier Goizueta `<jgoizueta@jazzfree.com>`
- Alan Karp `<karp@hp.com>`

Chapter 2

The Overall Structure

Processing a web requires three major steps:

1. Read the source, accumulating file names, fragment names, scraps, and lists of cross-references.
2. Reread the source, copying out to the documentation file, with protection and cross-reference information for all the scraps.
3. Traverse the list of files names. For each file name:
   (a) Dump all the defining scraps into a temporary file.
   (b) If the file already exists and is unchanged, delete the temporary file; otherwise, rename the temporary file.

2.1 Files

I have divided the program into several files for quicker recompilation during development.

`"global.h"` 10≡
⟨ *Include files* 11 ⟩
⟨ *Type declarations* 12a, ... ⟩
⟨ *Limits* 12b ⟩
⟨ *Global variable declarations* 16, ... ⟩
⟨ *Function prototypes* 25a, ... ⟩
⟨ *Operating System Dependencies* 15b ⟩◇

We'll need at least five of the standard system include files.

⟨ *Include files* 11 ⟩ ≡

```
/* #include <fcntl.h> */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <signal.h>
#include <locale.h>
◇
```

Fragment referenced in 10.

Defines: `exit` 15a, 21c, 28c, 29a, 52a, 84a, 89, 90, 91a, 92a, 97, 98, 100, 105a, 108a, 111, 124, 126, 127a, 128bc, 130, 131, 136b, 143b, `fclose` 40b, 70a, 83c, 84a, 85a, 91b, 115a, `FILE` 40b, 45b, 50b, 53, 54, 61, 62b, 64b, 67c, 70a, 75abc, 76, 79c, 81b, 83c, 85a, 87ab, 94ab, 106a, 113b, 115a, 145a, 146b, 148cd, 154, `fopen` 40b, 70a, 84a, 85a, 90, 92a, 115a, `fprintf` 19, 21bc, 25b, 26, 28c, 29a, 32d, 38, 39a, 40b, 41b, 44a, 46a, 49b, 50a, 52a, 53, 58ab, 59, 61, 62c, 68, 70a, 72b, 74c, 78a, 80a, 82a, 83c, 84a, 85b, 89, 90, 91a, 92a, 94ac, 97, 98, 100, 105a, 108ab, 110c, 111, 113ab, 120a, 124, 125ab, 126, 127a, 128bc, 130, 131, 136b, 143b, 145c, 147a, 149abef, `fputs` 32d, 33b, 35, 38, 39a, 41a, 43, 44ab, 45b, 46abcd, 47, 48ad, 49abc, 50ab, 53, 54, 56a, 57a, 58ab, 59, 60, 61, 62ac, 63ab, 65b, 66ab, 67b, 68, 72b, 73abcd, 74abc, 75abc, 76, 78abc, 79ab, 80abd, 81a, 82a, 94a, 107, 110c, 112a, 145bc, 146ac, 147a, `getc` 85a, 87c, 89, 90, 91ab, 92a, `getenv` 21b, `isgraph` 91a, 124, 130, `islower` 121a, `isspace` 48a, 100, 124, 125a, 126, 128c, `isupper` Never used, `malloc` 64b, 136b, 152c, `putc` 33bc, 41c, 42a, 54, 55c, 59, 61, 62c, 63a, 65b, 66ab, 68, 70b, 71, 76, 78a, 80ac, 94a, 107, 108d, 109bc, 111, 112abc, `remove` 84b, 85a, `setlocale` 21b, `size_t` 152ac, `stderr` 19, 21bc, 25b, 26, 28c, 29a, 40b, 49b, 50a, 52a, 59, 61, 70a, 74c, 78a, 83c, 84a, 85b, 89, 90, 91a, 92a, 94c, 97, 98, 100, 105a, 108a, 111, 120a, 124, 125ab, 126, 127a, 128bc, 130, 131, 136b, 143b, 149bef, `strlen` 33b, 118b, 119a, 138b, 148a, `toupper` 121a.

I like to use `TRUE` and `FALSE` in my code. I'd use an `enum` here, except that some systems seem to provide definitions of `TRUE` and `FALSE` by default. The following code seems to work on all the local systems.

⟨ *Type declarations* 12a ⟩ ≡
```
#ifndef FALSE
#define FALSE 0
#endif
#ifndef TRUE
#define TRUE 1
#endif
◇
```
Fragment defined by 12a, 83b, 116d, 117a, 127c, 129b, 132d, 144c, 150c.
Fragment referenced in 10.
Defines: `FALSE` 16, 17a, 19, 20abc, 21a, 41c, 42a, 46d, 48c, 50b, 61, 66a, 67a, 68, 79a, 89, 92a, 111, 115a, 120b, 123, 125a, 144a, 145c, 147ab, 148a, `TRUE` 16, 17a, 19, 21a, 24b, 38, 39a, 42a, 46b, 47, 50b, 61, 66a, 67a, 68, 73c, 78c, 89, 94bc, 111, 120b, 123, 125a, 144a, 145c, 147ab, 148a.

Here we define the maximum length for names (of fragments etc).

⟨ *Limits* 12b ⟩ ≡
```
#ifndef MAX_NAME_LEN
#define MAX_NAME_LEN 1024
#endif
◇
```
Fragment referenced in 10.
Defines: `MAX_NAME_LEN` 100, 105a, 124, 126, 130, 132a, 143b, 148b.

2.1.1 The Main Files

The code is divided into four main files (introduced here) and five support files (introduced in the next section). The file `main.c` will contain the driver for the whole program (see Section 2.2).

`"main.c"` 12c≡
```
#include "global.h"
◇
```
File defined by 12c, 15a.

The first pass over the source file is contained in `pass1.c`. It handles collection of all the file names, fragments names, and scraps (see Section 2.3).

`"pass1.c"` 12d≡
```
#include "global.h"
◇
```
File defined by 12d, 25b.

The `.tex` file is created during a second pass over the source file. The file `latex.c` contains the code controlling the construction of the `.tex` file (see Section 2.4).

`"latex.c"` 12e≡
```
#include "global.h"
static int scraps = 1;
int extra_scraps;
◇
```
File defined by 12e, 40ab, 45b, 50b, 51, 52ac, 53, 54, 55a, 61, 62b, 64ab, 67ac.
Uses: `scraps` 93a.

The file `html.c` contains the code controlling the construction of the `.tex` file appropriate for use with LaTeX2HTML (see Section 2.5).

"`html.c`" 13a≡

```
#include "global.h"
static int scraps = 1;
int extra_scraps;
◇
```

File defined by 13a, 69b, 70a, 75abc, 76, 79c, 81b.
Uses: `scraps` 93a.

The code controlling the creation of the output files is in `output.c` (see Section 2.6).

"`output.c`" 13b≡

```
#include "global.h"
◇
```

File defined by 13b, 83a.

2.1.2 Support Files

The support files contain a variety of support routines used to define and manipulate the major data abstractions. The file `input.c` holds all the routines used for referring to source files (see Section 3.1).

```
"input.c" 14a≡
    #include "global.h"
    ◇
```

File defined by 14a, 87abc, 88, 92a.

Creation and lookup of scraps is handled by routines in `scraps.c` (see Section 3.2).

```
"scraps.c" 14b≡
    #include "global.h"
    ◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.

The handling of file names and fragment names is detailed in `names.c` (see Section 3.3).

```
"names.c" 14c≡
    #include "global.h"
    ◇
```

File defined by 14c, 118b, 119ab, 120b, 121a, 122, 124, 126, 129ac, 130, 133c, 134a.

Memory allocation and deallocation is handled by routines in `arena.c` (see Section 3.6).

```
"arena.c" 14d≡
    #include "global.h"
    ◇
```

File defined by 14d, 151cd, 152a, 153.

Finally, for best portability, I seem to need a file containing (useless!) definitions of all the global variables.

```
"global.c" 14e≡
    #include "global.h"
    ⟨ Operating System Dependencies 15b ⟩
    ⟨ Global variable definitions 17a, ... ⟩
    ◇
```

File defined by 14e, 150b.

2.2 The Main Routine

The main routine is quite simple in structure. It wades through the optional command-line arguments, then handles any files listed on the command line.

"main.c" 15a≡

```
#include <stdlib.h>
int main(argc, argv)
     int argc;
     char **argv;
{
  int arg = 1;
  ⟨ Interpret command-line arguments 18b, ... ⟩
  ⟨ Set locale information 21b ⟩
  initialise_delimit_scrap_array();
  ⟨ Process the remaining arguments (file names) 21c ⟩
  exit(0);
}
◇
```

File defined by 12c, 15a.
Defines: `main` Never used.
Uses: `exit` 11.

We only have two major operating system dependencies; the separators for file names, and how to set environment variables. For now we assume the latter can be accomplished via `putenv()` in `stdlib.h`.

⟨ *Operating System Dependencies* 15b ⟩ ≡

```
#if defined(VMS)
#define PATH_SEP(c) (c==']'||c==':')
#define PATH_SEP_CHAR ""
#define DEFAULT_PATH ""
#elif defined(MSDOS)
#define PATH_SEP(c) (c=='\\')
#define PATH_SEP_CHAR "\\"
#define DEFAULT_PATH "."
#else
#define PATH_SEP(c) (c=='/')
#define PATH_SEP_CHAR "/"
#define DEFAULT_PATH "."
#endif
◇
```

Fragment referenced in 10, 14e.

2.2.1 Command-Line Arguments

There are numerous possible command-line arguments:

`-t` Suppresses generation of the `.tex` file.

`-o` Suppresses generation of the output files.

`-d` list dangling identifier references in indexes.

`-c` Forces output files to overwrite old files of the same name without comparing for equality first.

`-v` The verbose flag. Forces output of progress reports.

`-n` Forces sequential numbering of scraps (instead of page numbers).

`-s` Doesn't print list of scraps making up file at end of each scrap.

`-x` Include cross-reference numbers in scrap comments.

`-p` *path* Prepend *path* to the filenames for all the output files.

`-h` *options* Provide options for the hyperref package.

`-r` Turn on hyperlinks. You must include the —usepackage— options in the text.

There are two ways to get hyper-links into your documentation. With one, the `-r` flag simply arranges for fragment cross-references to be hyperlinks. You have to provide all the rest of the `hyperref` material yourself. The `-h` flag does more work, but requires you provide the `hyperref` options on the command line. You would use the first if you know you will use the same tools all the time. You would use the second if you have to distribute the nuweb file and use the build system to provide the `hyperref` options.

The `-h` option is used as follows. If you want hyperlinks in your document give a string to this option which is a valid option for the `hyperref` package. For example, `"dvips,colorlinks=true"` sets it to use the `dvips` driver and mark the links in colour. For more information about these options, see the `hyperref` documentation.

Then in your document you put the command `\NWuseHyperlinks` immadiately after your `usepackage` commands. If you do both of these you will get hyperlinks. If you miss out either then nothing interesting happens.

Global flags are declared for each of the arguments.

⟨ *Global variable declarations* 16 ⟩ ≡

```
extern int tex_flag;      /* if FALSE, don't emit the documentation file */
extern int html_flag;     /* if TRUE, emit HTML instead of LaTeX scraps. */
extern int output_flag;   /* if FALSE, don't emit the output files */
extern int compare_flag;  /* if FALSE, overwrite without comparison */
extern int verbose_flag;  /* if TRUE, write progress information */
extern int number_flag;   /* if TRUE, use a sequential numbering scheme */
extern int scrap_flag;    /* if FALSE, don't print list of scraps */
extern int dangling_flag;    /* if FALSE, don't print dangling identifiers */
extern int xref_flag; /* If TRUE, print cross-references in scrap comments */
extern int prepend_flag;  /* If TRUE, prepend a path to the output file names */
extern char * dirpath;    /* The prepended directory path */
extern char * path_sep;   /* How to join path to filename */
extern int listings_flag;   /* if TRUE, use listings package for scrap formatting */
extern int version_info_flag; /* If TRUE, set up version string */
extern char *  version_string; /* What to print for @v */
extern int hyperref_flag; /* Are we preparing for hyperref
                             package. */
extern int hyperopt_flag; /* Are we preparing for hyperref options */
extern char * hyperoptions; /* The options to pass to the
                               hyperref package */
extern int includepath_flag; /* Do we have an include path? */
extern struct incl{char * name; struct incl * next;} * include_list;
                       /* The list of include paths */
◇
```

Fragment defined by 16, 17bd, 27c, 32c, 48b, 86b, 95a, 117b, 151a.
Fragment referenced in 10.
Defines: `compare_flag` 17a, 19, 84b, `dangling_flag` 17a, 19, 68, `html_flag` 17a, 23, 24b, `hyperoptions` 17a, 21a, 41ab, `hyperopt_flag` 17a, 19, 21a, `hyperref_flag` 17a, 19, 21a, 41a, `incl` 17a, 20b, 24a, 90, `includepath_flag` 17a, 19, 20b, `number_flag` 17a, 19, 24b, 115a, `output_flag` 17a, 19, 24b, `scrap_flag` 17a, 19, 44a, `tex_flag` 17a, 19, 24b, 25b, 111, `verbose_flag` 17a, 19, 25b, 40b, 70a, 83c, `version_info_flag` 17a, 19, 20c, `version_string` 17a, 20c, 43, 57a, `xref_flag` 17a, 19, 112ac.
Uses: `FALSE` 12a, `scraps` 93a, `TRUE` 12a.

The flags are all initialized for correct default behavior.

⟨ *Global variable definitions* 17a ⟩ ≡

```
int tex_flag = TRUE;
int html_flag = FALSE;
int output_flag = TRUE;
int compare_flag = TRUE;
int verbose_flag = FALSE;
int number_flag = FALSE;
int scrap_flag = TRUE;
int dangling_flag = FALSE;
int xref_flag = FALSE;
int prepend_flag = FALSE;
char * dirpath = DEFAULT_PATH; /* Default directory path */
char * path_sep = PATH_SEP_CHAR;
int listings_flag = FALSE;
int version_info_flag = FALSE;
char default_version_string[] = "no version";
char *  version_string = default_version_string;
int hyperref_flag = FALSE;
int hyperopt_flag = FALSE;
char * hyperoptions = "";
int includepath_flag = FALSE; /* Do we have an include path? */
struct incl * include_list = NULL;
                        /* The list of include paths */
◇
```

Fragment defined by 17ac, 18a, 27d, 86c, 95b, 117c.
Fragment referenced in 14e.
Uses: `compare_flag` 16, `dangling_flag` 16, `FALSE` 12a, `html_flag` 16, `hyperoptions` 16, `hyperopt_flag` 16, `hyperref_flag` 16, `incl` 16, `includepath_flag` 16, `number_flag` 16, `output_flag` 16, `scrap_flag` 16, `tex_flag` 16, `TRUE` 12a, `verbose_flag` 16, `version_info_flag` 16, `version_string` 16, `xref_flag` 16.

A global variable `nw_char` will be used for the nuweb meta-character, which by default will be @.

⟨ *Global variable declarations* 17b ⟩ ≡

```
extern int nw_char;
◇
```

Fragment defined by 16, 17bd, 27c, 32c, 48b, 86b, 95a, 117b, 151a.
Fragment referenced in 10.
Defines: `nw_char` 17c, 25c, 26, 28c, 29a, 30b, 32b, 33a, 34b, 36, 37, 38, 39a, 41c, 42a, 44a, 51, 53, 54, 55a, 56a, 57d, 58a, 62c, 68, 70b, 71, 76, 77, 87c, 89, 92a, 97, 98, 99a, 100, 101cd, 105a, 107, 109ac, 110c, 111, 113ab, 114a, 120a, 124, 126, 127ab, 128c, 130, 131, 132a, 143ab, 145c, 147ab, 148b.

⟨ *Global variable definitions* 17c ⟩ ≡

```
int nw_char='@';
◇
```

Fragment defined by 17ac, 18a, 27d, 86c, 95b, 117c.
Fragment referenced in 14e.
Defines: `nw_char` 17b, 25c, 26, 28c, 29a, 30b, 32b, 33a, 34b, 36, 37, 38, 39a, 41c, 42a, 44a, 51, 53, 54, 55a, 56a, 57d, 58a, 62c, 68, 70b, 71, 76, 77, 87c, 89, 92a, 97, 98, 99a, 100, 101cd, 105a, 107, 109ac, 110c, 111, 113ab, 114a, 120a, 124, 126, 127ab, 128c, 130, 131, 132a, 143ab, 145c, 147ab, 148b.

We save the invocation name of the command in a global variable `command_name` for use in error messages.

⟨ *Global variable declarations* 17d ⟩ ≡

```
extern char *command_name;
◇
```

Fragment defined by 16, 17bd, 27c, 32c, 48b, 86b, 95a, 117b, 151a.
Fragment referenced in 10.
Defines: `command_name` 18ab, 19, 21c, 26, 28c, 29a, 40b, 50a, 59, 61, 70a, 74c, 78a, 84a, 85b, 89, 90, 91a, 92a, 94c, 97, 98, 100, 105a, 111, 120a, 124, 125ab, 126, 127a, 128bc, 130, 131, 143b.

⟨ *Global variable definitions* 18a ⟩ ≡

```
char *command_name = NULL;
◇
```

Fragment defined by 17ac, 18a, 27d, 86c, 95b, 117c.
Fragment referenced in 14e.
Uses: `command_name` 17d.

The invocation name is conventionally passed in `argv[0]`.

⟨ *Interpret command-line arguments* 18b ⟩ ≡

```
command_name = argv[0];
◇
```

Fragment defined by 18bc.
Fragment referenced in 15a.
Uses: `command_name` 17d.

We need to examine the remaining entries in `argv`, looking for command-line arguments.

⟨ *Interpret command-line arguments* 18c ⟩ ≡

```
while (arg < argc) {
  char *s = argv[arg];
  if (*s++ == '-') {
    ⟨ Interpret the argument string s 19 ⟩
    arg++;
    ⟨ Perhaps get the prepend path 20a ⟩
    ⟨ Perhaps get the version info string 20c ⟩
    ⟨ Perhaps get the hyperref options 21a ⟩
    ⟨ Perhaps add an include path 20b ⟩
  }
  else break;
}◇
```

Fragment defined by 18bc.
Fragment referenced in 15a.

Several flags can be stacked behind a single minus sign; therefore, we've got to loop through the string, handling them all. If this flag requires an argument we skip to getting its argument straight away. This allows arguments to butt up to their flags and also avoids ambiguity about which value goes with which flag.

⟨ *Interpret the argument string* s 19 ⟩ ≡

```
{
  char c = *s++;
  while (c) {
    switch (c) {
      case 'c': compare_flag = FALSE;
                break;
      case 'd': dangling_flag = TRUE;
                break;
      case 'h': hyperopt_flag = TRUE;
                goto HasValue;
      case 'I': includepath_flag = TRUE;
                goto HasValue;
      case 'l': listings_flag = TRUE;
                break;
      case 'n': number_flag = TRUE;
                break;
      case 'o': output_flag = FALSE;
                break;
      case 'p': prepend_flag = TRUE;
                goto HasValue;
      case 'r': hyperref_flag = TRUE;
                break;
      case 's': scrap_flag = FALSE;
                break;
      case 't': tex_flag = FALSE;
                break;
      case 'v': verbose_flag = TRUE;
                break;
      case 'V': version_info_flag = TRUE;
                goto HasValue;
      case 'x': xref_flag = TRUE;
                break;
      default:  fprintf(stderr, "%s: unexpected argument ignored.  ",
                        command_name);
                fprintf(stderr, "Usage is: %s [-cdnostvx] "
                      "[-I path] [-V version] "
                      "[-h options] [-p path] file...\n",
                        command_name);
                break;
    }
    c = *s++;
  }
HasValue:;
}◇
```

Fragment referenced in 18c.

Uses: command_name 17d, compare_flag 16, dangling_flag 16, FALSE 12a, fprintf 11, hyperopt_flag 16, hyperref_flag 16, includepath_flag 16, number_flag 16, output_flag 16, scrap_flag 16, stderr 11, tex_flag 16, TRUE 12a, verbose_flag 16, version_info_flag 16, xref_flag 16.

⟨ *Perhaps get the prepend path* 20a ⟩ ≡

```
if (prepend_flag)
{
  if (*s == '\0')
    s = argv[arg++];
  dirpath = s;
  prepend_flag = FALSE;
}
◇
```

Fragment referenced in 18c.
Uses: FALSE 12a.

⟨ *Perhaps add an include path* 20b ⟩ ≡

```
if (includepath_flag)
{
   struct incl * le
      = (struct incl *)arena_getmem(sizeof(struct incl));
   struct incl ** p = &include_list;

   if (*s == '\0')
     s = argv[arg++];
   le->name = save_string(s);
   le->next = NULL;
   while (*p != NULL)
      p = &((*p)->next);
   *p = le;
   includepath_flag = FALSE;
}
◇
```

Fragment referenced in 18c.
Uses: arena_getmem 152a, FALSE 12a, incl 16, includepath_flag 16, save_string 119a.

⟨ *Perhaps get the version info string* 20c ⟩ ≡

```
if (version_info_flag)
{
   if (*s == '\0')
     s = argv[arg++];
   version_string = s;
   version_info_flag = FALSE;
}
◇
```

Fragment referenced in 18c.
Uses: FALSE 12a, version_info_flag 16, version_string 16.

⟨ *Perhaps get the hyperref options* 21a ⟩ ≡

```
if (hyperopt_flag)
{
  if (*s == '\0')
    s = argv[arg++];
  hyperoptions = s;
  hyperopt_flag = FALSE;
  hyperref_flag = TRUE;
}
◇
```

Fragment referenced in 18c.
Uses: FALSE 12a, hyperoptions 16, hyperopt_flag 16, hyperref_flag 16, TRUE 12a.

In order to be able to process files in foreign languages we set the locale information. `isgraph()` and friends need this (see Section 3.3). Try to read `LC_CTYPE` from the environment. Use `LC_ALL` if that fails. Don't set the locale if reading `LC_ALL` fails, too. Print a warning if setting the program's locale fails.

⟨ *Set locale information* 21b ⟩ ≡

```
{
  /* try to get locale information */
  char *s=getenv("LC_CTYPE");
  if (s==NULL) s=getenv("LC_ALL");

  /* set it */
  if (s!=NULL)
    if(setlocale(LC_CTYPE, s)==NULL)
      fprintf(stderr, "Setting locale failed\n");
}
◇
```

Fragment referenced in 15a.
Uses: fprintf 11, getenv 11, setlocale 11, stderr 11.

2.2.2 File Names

We expect at least one file name. While a missing file name might be ignored without causing any problems, we take the opportunity to report the usage convention.

⟨ *Process the remaining arguments (file names)* 21c ⟩ ≡

```
{
  if (arg >= argc) {
    fprintf(stderr, "%s: expected a file name.  ", command_name);
    fprintf(stderr, "Usage is: %s [-cnotv] [-p path] file-name...\n", command_name);
    exit(-1);
  }
  do {
    ⟨ Handle the file name in argv[arg] 22 ⟩
    arg++;
  } while (arg < argc);
}◇
```

Fragment referenced in 15a.
Uses: command_name 17d, exit 11, fprintf 11, stderr 11.

The code to handle a particular file name is rather more tedious than the actual processing of the file. A file name may be an arbitrarily complicated path name, with an optional extension. If no extension is present, we add `.w` as a default. The extended path name will be kept in a local variable `source_name`. The resulting documentation file will be written in the current directory; its name will be kept in the variable `tex_name`.

⟨ *Handle the file name in* `argv[arg]` 22 ⟩ ≡

```
{
  char source_name[FILENAME_MAX];
  char tex_name[FILENAME_MAX];
  char aux_name[FILENAME_MAX];
  ⟨ Build source_name and tex_name 23 ⟩
  ⟨ Process a file 24b ⟩
}◇
```

Fragment referenced in 21c.
Uses: `source_name` 86b.

I bump the pointer `p` through all the characters in `argv[arg]`, copying all the characters into `source_name` (via the pointer `q`).

At each slash, I update `trim` to point just past the slash in `source_name`. The effect is that `trim` will point at the file name without any leading directory specifications.

The pointer `dot` is made to point at the file name extension, if present. If there is no extension, we add `.w` to the source name. In any case, we create the `tex_name` from `trim`, taking care to get the correct extension. The `html_flag` is set in this scrap.

⟨ *Build* `source_name` *and* `tex_name` 23 ⟩ ≡

```
{
  char *p = argv[arg];
  char *q = source_name;
  char *trim = q;
  char *dot = NULL;
  char c = *p++;
  while (c) {
    *q++ = c;
    if (PATH_SEP(c)) {
      trim = q;
      dot = NULL;
    }
    else if (c == '.')
      dot = q - 1;
    c = *p++;
  }
  ⟨ Add the source path to the include path list 24a ⟩
  *q = '\0';
  if (dot) {
    *dot = '\0'; /* produce HTML when the file extension is ".hw" */
    html_flag = dot[1] == 'h' && dot[2] == 'w' && dot[3] == '\0';
    sprintf(tex_name, "%s%s%s.tex", dirpath, path_sep, trim);
    sprintf(aux_name, "%s%s%s.aux", dirpath, path_sep, trim);
    *dot = '.';
  }
  else {
    sprintf(tex_name, "%s%s%s.tex", dirpath, path_sep, trim);
    sprintf(aux_name, "%s%s%s.aux", dirpath, path_sep, trim);
    *q++ = '.';
    *q++ = 'w';
    *q = '\0';
  }
}◇
```

Fragment referenced in 22.
Uses: `html_flag` 16, `source_name` 86b.

If the source file has a directory part we add that directory to the end of the search path so that the path of last resort is the same as the source path, not, as one person put it, something irrelevant like the directory nuweb was invoked from.

⟨ *Add the source path to the include path list* 24a ⟩ ≡

```
if (trim != source_name) {
   struct incl * le
      = (struct incl *)arena_getmem(sizeof(struct incl));
   struct incl ** p = &include_list;
   char sv = *trim;

   *trim = '\0';
   le->name = save_string(source_name);
   le->next = NULL;
   while (*p != NULL)
      p = &((*p)->next);
   *p = le;
   *trim = sv;
}
```

◇

Fragment referenced in 23.
Uses: `arena_getmem` 152a, `incl` 16, `save_string` 119a, `source_name` 86b.

Now that we're finally ready to process a file, it's not really too complex. We bundle most of the work into four routines `pass1` (see Section 2.3), `write_tex` (see Section 2.4), `write_html` (see Section 2.5), and `write_files` (see Section 2.6). After we're finished with a particular file, we must remember to release its storage (see Section 3.6). The sequential numbering of scraps is forced when generating HTML.

⟨ *Process a file* 24b ⟩ ≡

```
{
  pass1(source_name);
  current_sector = 1;
  prev_sector = 1;
  if (tex_flag) {
    if (html_flag) {
      int saved_number_flag = number_flag;
      number_flag = TRUE;
      collect_numbers(aux_name);
      write_html(source_name, tex_name, 0/*Dummy */);
      number_flag = saved_number_flag;
    }
    else {
      collect_numbers(aux_name);
      write_tex(source_name, tex_name);
    }
  }
  if (output_flag)
    write_files(file_names);
  arena_free();
}◇
```

Fragment referenced in 22.
Uses: `arena_free` 153, `collect_numbers` 115a, `current_sector` 27c, `file_names` 117b, `html_flag` 16, `number_flag` 16, `output_flag` 16, `pass1` 25b, `prev_sector` 27c, `source_name` 86b, `tex_flag` 16, `TRUE` 12a, `write_files` 83a, `write_html` 70a, `write_tex` 40b.

2.3 Pass One

During the first pass, we scan the file, recording the definitions of each fragment and file and accumulating all the scraps.

⟨ *Function prototypes* 25a ⟩ ≡
```
extern void pass1();
◇
```
Fragment defined by 25a, 39b, 52b, 55b, 69a, 82b, 86a, 93b, 102b, 114b, 118a, 128d, 139a, 148d, 151b.
Fragment referenced in 10.
Uses: pass1 25b.

The routine pass1 takes a single argument, the name of the source file. It opens the file, then initializes the scrap structures (see Section 3.2) and the roots of the file-name tree, the fragment-name tree, and the tree of user-specified index entries (see Section 3.3). After completing all the necessary preparation, we make a pass over the file, filling in all our data structures. Next, we seach all the scraps for references to the user-specified index entries. Finally, we must reverse all the cross-reference lists accumulated while scanning the scraps.

"pass1.c" 25b≡
```
void pass1(file_name)
     char *file_name;
{
  if (verbose_flag)
    fprintf(stderr, "reading %s\n", file_name);
  source_open(file_name);
  init_scraps();
  macro_names = NULL;
  file_names = NULL;
  user_names = NULL;
  ⟨ Scan the source file, looking for at-sequences 25c ⟩
  if (tex_flag)
    search();
  ⟨ Reverse cross-reference lists 34d ⟩
}
◇
```
File defined by 12d, 25b.
Defines: pass1 24b, 25a, 56a, 77.
Uses: file_names 117b, fprintf 11, init_scraps 93c, macro_names 117b, search 139b, source_open 92a, stderr 11, tex_flag 16, user_names 117b, verbose_flag 16.

The only thing we look for in the first pass are the command sequences. All ordinary text is skipped entirely.

⟨ *Scan the source file, looking for at-sequences* 25c ⟩ ≡
```
{
  int c = source_get();
  while (c != EOF) {
    if (c == nw_char)
      ⟨ Scan at-sequence 26 ⟩
    c = source_get();
  }
}◇
```
Fragment referenced in 25b.
Uses: nw_char 17bc, source_get 87c.

Only four of the at-sequences are interesting during the first pass. We skip past others immediately; warning if unexpected sequences are discovered.

⟨ *Scan at-sequence* 26 ⟩ ≡

```
{
  char quoted = 0;

  c = source_get();
  switch (c) {
    case 'r':
          c = source_get();
          nw_char = c;
          update_delimit_scrap();
          break;
    case 'O':
    case 'o': ⟨ Build output file definition 27e ⟩
              break;
    case 'Q':
    case 'q': quoted = 1;
    case 'D':
    case 'd': ⟨ Build fragment definition 28a ⟩
              break;
    case 's':
              ⟨ Step to next sector 27a ⟩
              break;
    case 'S':
              ⟨ Close the current sector 27b ⟩
              break;
    case '(':
    case '[':
    case '{': ⟨ Skip over an in-text scrap 28c ⟩
              break;
    case 'c': ⟨ Collect a block comment 29b ⟩
              break;
    case 'x':
    case 'v':
    case 'u':
    case 'm':
    case 'f': /* ignore during this pass */
              break;
    default:  if (c==nw_char) /* ignore during this pass */
                break;
              fprintf(stderr,
                      "%s: unexpected %c sequence ignored (%s, line %d)\n",
                      command_name, nw_char, source_name, source_line);
              break;
  }
}◇
```

Fragment referenced in 25c.
Uses: `command_name` 17d, `fprintf` 11, `nw_char` 17bc, `source_get` 87c, `source_line` 86b, `source_name` 86b, `stderr` 11, `update_delimit_scrap` 55a.

⟨ *Step to next sector* 27a ⟩ ≡

```
prev_sector += 1;
current_sector = prev_sector;
c = source_get();
◇
```

Fragment referenced in 26, 42a.
Uses: `current_sector` 27c, `prev_sector` 27c, `source_get` 87c.

⟨ *Close the current sector* 27b ⟩ ≡

```
current_sector = 1;
c = source_get();
◇
```

Fragment referenced in 26, 42a.
Uses: `current_sector` 27c, `source_get` 87c.

⟨ *Global variable declarations* 27c ⟩ ≡

```
unsigned char current_sector;
unsigned char prev_sector;
◇
```

Fragment defined by 16, 17bd, 27c, 32c, 48b, 86b, 95a, 117b, 151a.
Fragment referenced in 10.
Defines: `current_sector` 24b, 27abd, 63b, 67b, 96c, 100, 126, 130, `prev_sector` 24b, 27ad.

⟨ *Global variable definitions* 27d ⟩ ≡

```
unsigned char current_sector = 1;
unsigned char prev_sector = 1;
◇
```

Fragment defined by 17ac, 18a, 27d, 86c, 95b, 117c.
Fragment referenced in 14e.
Uses: `current_sector` 27c, `prev_sector` 27c.

2.3.1 Accumulating Definitions

There are three steps required to handle a definition:

1. Build an entry for the name so we can look it up later.
2. Collect the scrap and save it in the table of scraps.
3. Attach the scrap to the name.

We go through the same steps for both file names and fragment names.

⟨ *Build output file definition* 27e ⟩ ≡

```
{
  Name *name = collect_file_name(); /* returns a pointer to the name entry */
  int scrap = collect_scrap();      /* returns an index to the scrap */
  ⟨ Add scrap to name's definition list 28b ⟩
}◇
```

Fragment referenced in 26.
Uses: `collect_file_name` 124, `collect_scrap` 96b, `Name` 117a.

⟨ *Build fragment definition* 28a ⟩ ≡

```
{
  Name *name = collect_macro_name();
  int scrap = collect_scrap();
  ⟨ Add scrap to name's definition list 28b ⟩
}◇
```

Fragment referenced in 26.
Uses: collect_macro_name 126, collect_scrap 96b, Name 117a.

Since a file or fragment may be defined by many scraps, we maintain them in a simple linked list. The list is actually built in reverse order, with each new definition being added to the head of the list.

⟨ *Add* scrap *to* name*'s definition list* 28b ⟩ ≡

```
{
  Scrap_Node *def = (Scrap_Node *) arena_getmem(sizeof(Scrap_Node));
  def->scrap = scrap;
  def->quoted = quoted;
  def->next = name->defs;
  name->defs = def;
}◇
```

Fragment referenced in 27e, 28a.
Uses: arena_getmem 152a, Scrap_Node 116d.

⟨ *Skip over an in-text scrap* 28c ⟩ ≡

```
{
   int c;
   while ((c = source_get()) != EOF) {
      if (c == nw_char)
         ⟨ Skip over at-sign or go to skipped 29a ⟩
   }
   fprintf(stderr, "%s: unexpected EOF in text at (%s, %d)\n",
                    command_name, source_name, source_line);
   exit(-1);

skipped:  ;
}
◇
```

Fragment referenced in 26.
Uses: command_name 17d, exit 11, fprintf 11, nw_char 17bc, source_get 87c, source_line 86b, source_name 86b, stderr 11.

⟨ *Skip over at-sign or go to skipped* 29a ⟩ ≡

```
{
   c = source_get();
   switch (c) {
     case '}': case ']': case ')':
        goto skipped;
     case 'x': case '|': case ',': case '<':
     case '>': case '%': case '1': case '2':
     case '3': case '4': case '5': case '6':
     case '7': case '8': case '9': case '_':
     case 'f': case '#': case '+': case '-':
     case 'v': case '*': case 'c': case '\'':
        break;
     default:
        if (c != nw_char) {
           fprintf(stderr, "%s: unexpected %c%c in text at (%s, %d)\n",
                           command_name, nw_char, c, source_name, source_line);
           exit(-1);
        }
        break;
   }
}
◇
```

Fragment referenced in 28c.
Uses: `command_name` 17d, `exit` 11, `fprintf` 11, `nw_char` 17bc, `source_get` 87c, `source_line` 86b, `source_name` 86b, `stderr` 11.

2.3.2 Block Comments

⟨ *Collect a block comment* 29b ⟩ ≡

```
{
   char * p = blockBuff;
   char * e = blockBuff + (sizeof(blockBuff)/sizeof(blockBuff[0])) - 1;

   ⟨ Skip whitespace 30a ⟩
   while (p < e)
   {
      ⟨ Add one char to the block buffer 30b ⟩
   }
   if (p == e)
   {
      ⟨ Skip to the next nw-char 32b ⟩
   }
   *p = '\000';
}
◇
```

Fragment referenced in 26.
Uses: `blockBuff` 32c.

⟨ *Skip whitespace* 30a ⟩ ≡

```
while (source_peek == ' '
       || source_peek == '\t'
       || source_peek == '\n')
   (void)source_get();
```

◇

Fragment referenced in 29b.
Uses: `source_get` 87c, `source_peek` 87c.

⟨ *Add one char to the block buffer* 30b ⟩ ≡

```
int c = source_get();

if (c == nw_char)
{
   ⟨ Add an at character to the block or break 31a ⟩
}
else if (c == EOF)
{
   source_ungetc(&c);
   break;
}
else
{
   ⟨ Add any other character to the block 31b ⟩
}
```

◇

Fragment referenced in 29b.
Uses: `nw_char` 17bc, `source_get` 87c, `source_ungetc` 88.

⟨ *Add an at character to the block or break* 31a ⟩ ≡

```
int cc = source_peek;

if (cc == 'c')
{
   do
      c = source_get();
   while (c <= ' ');

   break;
}
else if (cc == 'd'
         || cc == 'D'
         || cc == 'q'
         || cc == 'Q'
         || cc == 'o'
         || cc == 'O'
         || cc == EOF)
{
   source_ungetc(&c);
   break;
}
else
{
   *p++ = c;
   *p++ = source_get();
}
◇
```

Fragment referenced in 30b.
Uses: `source_get` 87c, `source_peek` 87c, `source_ungetc` 88.

⟨ *Add any other character to the block* 31b ⟩ ≡

```
   ⟨ Perhaps skip white-space 32a ⟩
   *p++ = c;
◇
```

Fragment referenced in 30b.

⟨ *Perhaps skip white-space* 32a ⟩ ≡

```
if (c == ' ')
{
   while (source_peek == ' ')
      c = source_get();
}
if (c == '\n')
{
   if (source_peek == '\n')
   {
      do
         c = source_get();
      while (source_peek == '\n');
   }
   else
      c = ' ';
}
◇
```

Fragment referenced in 31b.
Uses: `source_get` 87c, `source_peek` 87c.

⟨ *Skip to the next nw-char* 32b ⟩ ≡

```
int c;

while ((c = source_get()), c != nw_char && c != EOF)/* Skip */
source_ungetc(&c);◇
```

Fragment referenced in 29b.
Uses: `nw_char` 17bc, `source_get` 87c, `source_ungetc` 88.

⟨ *Global variable declarations* 32c ⟩ ≡

```
char blockBuff[6400];
◇
```

Fragment defined by 16, 17bd, 27c, 32c, 48b, 86b, 95a, 117b, 151a.
Fragment referenced in 10.
Defines: `blockBuff` 29b, 33a.

We don't show block comments inside scraps in the TEX file, but we do show where they go.

⟨ *Show presence of a block comment* 32d ⟩ ≡

```
{
  fputs(delimit_scrap[scrap_type][1],file);
  fprintf(file, "\\hbox{\\sffamily\\slshape (Comment)}");
  fputs(delimit_scrap[scrap_type][0], file);
}
◇
```

Fragment referenced in 56a.
Uses: `fprintf` 11, `fputs` 11, `scrap_type` 52c, 54.

The block comment is presently in the block buffer. Here we copy it into the scrap whence we shall copy it into the output file.

⟨ *Include block comment in a scrap* 33a ⟩ ≡

```
{
   char * p = blockBuff;

   push(nw_char, &writer);
   do
   {
      push(c, &writer);
      c = *p++;
   } while (c != '\0');
}◇
```

Fragment referenced in 98.
Uses: `blockBuff` 32c, `nw_char` 17bc, `push` 95d.

⟨ *Copy block comment from scrap* 33b ⟩ ≡

```
{
   int bgn = indent + global_indent;
   int posn = bgn + strlen(comment_begin[comment_flag]);
   int i;

   ⟨ Perhaps put a delayed indent 112b ⟩
   c = pop(&reader);
   fputs(comment_begin[comment_flag], file);
   while (c != '\0')
   {
      ⟨ Move a word to the file 33c ⟩
      ⟨ If we break the line at this word 34a ⟩
      {
         putc('\n', file);
         for (i = 0; i < bgn ; i++)
            putc(' ', file);
         c = pop(&reader);
         if (c != '\0')
         {
            posn = bgn + strlen(comment_mid[comment_flag]);
            fputs(comment_mid[comment_flag], file);
         }
      }
   }
   fputs(comment_end[comment_flag], file);
}
◇
```

Fragment referenced in 109c.
Uses: `comment_begin` 125c, `comment_end` 125c, `comment_mid` 125c, `fputs` 11, `pop` 102c, `putc` 11, `strlen` 11.

⟨ *Move a word to the file* 33c ⟩ ≡

```
do
{
   putc(c, file);
   posn += 1;
   c = pop(&reader);
} while (c > ' ');
◇
```

Fragment referenced in 33b.
Uses: `pop` 102c, `putc` 11.

⟨ *If we break the line at this word* 34a ⟩ ≡
```
if (c == '\n' || (c == ' ' && posn > 60))◇
```
Fragment referenced in 33b.

⟨ *Skip over a block comment* 34b ⟩ ≡
```
if (last == nw_char && c == 'c')
   while ((c = pop(reader.m)) != '\0')
      /* Skip */;
◇
```
Fragment referenced in 142.
Uses: nw_char 17bc, pop 102c.

⟨ *Begin or end a block comment* 34c ⟩ ≡
```
if (inBlock)
{
   ⟨ End block 46d ⟩
}
else
{
   ⟨ Start block 46b ⟩
}◇
```
Fragment referenced in 42a.

2.3.3 Fixing the Cross References

Since the definition and reference lists for each name are accumulated in reverse order, we take the time at the end of `pass1` to reverse them all so they'll be simpler to print out prettily. The code for `reverse_lists` appears in Section 3.3.

⟨ *Reverse cross-reference lists* 34d ⟩ ≡
```
{
  reverse_lists(file_names);
  reverse_lists(macro_names);
  reverse_lists(user_names);
}◇
```
Fragment referenced in 25b.
Uses: file_names 117b, macro_names 117b, reverse_lists 133c, user_names 117b.

2.3.4 Dealing with fragment parameters

Fragment parameters were added on later in nuweb's development. There still is not, for example, an index of fragment parameters. We need a data type to keep track of fragment parameters.

"scraps.c" 34e≡
```
typedef int *Parameters;
◇
```
File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: Parameters 37, 105a, 106a, 107.

When we are copying a scrap to the output, we check for an embedded parameter. If we find it we copy it. Otherwise it's an old-style parameter and we can then pull the nth string from the `Parameters` list when we see an `@1` `@2`, etc.

⟨ *Handle macro parameter substitution* 35 ⟩ ≡

```
case '1': case '2': case '3':
case '4': case '5': case '6':
case '7': case '8': case '9':
  {
    Arglist * args;
    Name * name;

    lookup(c - '1', inArgs, inParams, &name, &args);

    if (name == (Name *)1) {
      Embed_Node * q = (Embed_Node *)args;
      indent = write_scraps(file, spelling, q->defs,
                            global_indent + indent,
                            indent_chars, debug_flag,
                            tab_flag, indent_flag,
                            0,
                            q->args, inParams,
                            local_parameters, "");
    }
    else if (name != NULL) {
       int i, narg;
       char * p = name->spelling;
       Arglist *q = args;

      ⟨ Perhaps comment this macro 112a ⟩
      indent = write_scraps(file, spelling, name->defs,
                            global_indent + indent,
                            indent_chars, debug_flag,
                            tab_flag, indent_flag,
                            comment_flag, args, name->arg,
                            local_parameters, p);
    }
    else if (args != NULL) {
       if (delayed_indent) {
         ⟨ Put out the indent 108d ⟩
       }
       fputs((char *)args, file);
    }
    else if ( parameters && parameters[c - '1'] ) {
      Scrap_Node param_defs;
      param_defs.scrap = parameters[c - '1'];
      param_defs.next = 0;
      write_scraps(file, spelling, &param_defs,
                   global_indent + indent,
                   indent_chars, debug_flag,
                   tab_flag, indent_flag,
                   comment_flag, NULL, NULL, 0, "");
    } else if (delayed_indent) {
      ⟨ Put out the indent 108d ⟩
    }
  }
  break;
```

◇

Fragment referenced in 109c.
Uses: Arglist 129b, delayed_indent 106b, Embed_Node 132d, fputs 11, lookup 103b, Name 117a, Scrap_Node 116d, write_scraps 106a.

Now onto actually parsing fragment parameters from a call. We start off checking for fragment parameters, an @(sequence followed by parameters separated by @, sequences, and terminated by a @) sequence.

We collect separate scraps for each parameter, and write the scrap numbers down in the text. For example, if the file has:

```
@<foo @( param1 @, param2 @)@>
```

we actually make new scraps, say 10 and 11, for param1 and param2, and write in the collected scrap:

```
@<foo @(10@,11@)@>
```

⟨ *Save macro parameters* 36 ⟩ ≡

```
{
  int param_scrap;
  char param_buf[10];

  push(nw_char, &writer);
  push('(', &writer);
  do {
     param_scrap = collect_scrap();
     sprintf(param_buf, "%d", param_scrap);
     pushs(param_buf, &writer);
     push(nw_char, &writer);
     push(scrap_ended_with, &writer);
     add_to_use(name, current_scrap);
  } while( scrap_ended_with == ',' );
  do
    c = source_get();
  while( ' ' == c );
  if (c == nw_char) {
    c = source_get();
  }
  if (c != '>') {
    /* ZZZ print error */;
  }
}◇
```

Fragment referenced in 101c.
Uses: add_to_use 102a, collect_scrap 96b, nw_char 17bc, push 95d, pushs 96a, source_get 87c.

If we get inside, we have at least one parameter, which will be at the beginning of the parms buffer, and we prime the pump with the first character.

⟨ *Check for macro parameters* 37 ⟩ ≡

```
if (c == '(') {
  Parameters res = arena_getmem(10 * sizeof(int));
  int *p2 = res;
  int count = 0;
  int scrapnum;

  while( c && c != ')' ) {
    scrapnum = 0;
    c = pop(manager);
    while( '0' <= c && c <= '9' ) {
      scrapnum = scrapnum  * 10 + c - '0';
      c = pop(manager);
    }
    if ( c == nw_char ) {
      c = pop(manager);
    }
    *p2++ = scrapnum;
  }
  while (count < 10) {
    *p2++ = 0;
    count++;
  }
  while( c && c != nw_char ) {
      c = pop(manager);
  }
  if ( c == nw_char ) {
    c = pop(manager);
  }
  *parameters = res;
}
```

◇

Fragment referenced in 105b.
Uses: `arena_getmem` 152a, `nw_char` 17bc, `Parameters` 34e, `pop` 102c.

These are used in `write_tex` and `write_html` to output the argument list for a fragment.

⟨ *Format macro parameters* 38 ⟩ ≡

```
char sep;

sep = '(';
do {
  fputc(sep,file);

  fputs("{\\footnotesize ", file);
  write_single_scrap_ref(file, scraps + 1);
  fprintf(file, "\\label{scrap%d}\n", scraps + 1);
  fputs(" }", file);

  source_last = '{';
  copy_scrap(file, TRUE, NULL);

  ++scraps;

  sep = ',';
} while ( source_last != ')' && source_last != EOF );
fputs(" ) ",file);
do
  c = source_get();
while(c != nw_char && c != EOF);
if (c == nw_char) {
  c = source_get();
}
```

◇

Fragment referenced in 59, 61.
Uses: `copy_scrap` 54, 76, `fprintf` 11, `fputs` 11, `nw_char` 17bc, `scraps` 93a, `source_get` 87c, `source_last` 87c, `TRUE` 12a, `write_single_scrap_ref` 94b.

⟨ *Format HTML macro parameters* 39a ⟩ ≡

```
        char sep;

        sep = '(';
        fputs("\\begin{rawhtml}", file);
        do {

          fputc(sep,file);

          fprintf(file, "%d <A NAME=\"#nuweb%d\"></A>", scraps, scraps);

          source_last = '{';
          copy_scrap(file, TRUE);

          ++scraps;
          sep = ',';

        } while ( source_last != ')' && source_last != EOF );
        fputs(" ) ",file);
        do
          c = source_get();
        while(c != nw_char && c != EOF);
        if (c == nw_char) {
          c = source_get();
        }
        fputs("\\end{rawhtml}", file);
    ◇
```

Fragment referenced in 78a.
Uses: `copy_scrap` 54, 76, `fprintf` 11, `fputs` 11, `nw_char` 17bc, `scraps` 93a, `source_get` 87c, `source_last` 87c, `TRUE` 12a.

2.4 Writing the Latex File

The second pass (invoked via a call to `write_tex`) copies most of the text from the source file straight into a `.tex` file. Definitions are formatted slightly and cross-reference information is printed out.

Note that all the formatting is handled in this section. If you don't like the format of definitions or indices or whatever, it'll be in this section somewhere. Similarly, if someone wanted to modify nuweb to work with a different typesetting system, this would be the place to look.

⟨ *Function prototypes* 39b ⟩ ≡

```
    extern void write_tex();
    ◇
```

Fragment defined by 25a, 39b, 52b, 55b, 69a, 82b, 86a, 93b, 102b, 114b, 118a, 128d, 139a, 148d, 151b.
Fragment referenced in 10.
Uses: `write_tex` 40b.

We need a few local function declarations before we get into the body of `write_tex`.

"latex.c" 40a≡

```
static void copy_scrap();              /* formats the body of a scrap */
static void print_scrap_numbers();       /* formats a list of scrap numbers */
static void format_entry();              /* formats an index entry */
static void format_file_entry();         /* formats a file index entry */
static void format_user_entry();
static void write_arg();
static void write_literal();
static void write_ArglistElement();
◇
```

File defined by 12e, 40ab, 45b, 50b, 51, 52ac, 53, 54, 55a, 61, 62b, 64ab, 67ac.
Uses: copy_scrap 54, 76, format_entry 64b, 79c, format_file_entry 62b, format_user_entry 67c, 81b, print_scrap_numbers 50b, 75c, write_arg 45b, write_ArglistElement 61, write_literal 53.

The routine write_tex takes two file names as parameters: the name of the web source file and the name of the .tex output file.

"latex.c" 40b≡

```
void write_tex(file_name, tex_name, sector)
     char *file_name;
     char *tex_name;
     unsigned char sector;
{
  FILE *tex_file = fopen(tex_name, "w");
  if (tex_file) {
    if (verbose_flag)
      fprintf(stderr, "writing %s\n", tex_name);
    source_open(file_name);
    ⟨ Write LaTeX limbo definitions 41a ⟩
    ⟨ Copy source_file into tex_file 41c ⟩
    fclose(tex_file);
  }
  else
    fprintf(stderr, "%s: can't open %s\n", command_name, tex_name);
}
◇
```

File defined by 12e, 40ab, 45b, 50b, 51, 52ac, 53, 54, 55a, 61, 62b, 64ab, 67ac.
Defines: write_tex 24b, 39b.
Uses: command_name 17d, fclose 11, FILE 11, fopen 11, fprintf 11, source_open 92a, stderr 11, verbose_flag 16.

Now that the \NW... macros are used, it seems convenient to write default definitions for those macros so that source files need not define anything new. If a user wants to change any of the macros (to use hyperref or to write in some language other than english) he or she can redefine the commands.

⟨ *Write LaTeX limbo definitions* 41a ⟩ ≡

```
if (hyperref_flag) {
   fputs("\\newcommand{\\NWtarget}[2]{\\hypertarget{#1}{#2}}\n", tex_file);
   fputs("\\newcommand{\\NWlink}[2]{\\hyperlink{#1}{#2}}\n", tex_file);
} else {
   fputs("\\newcommand{\\NWtarget}[2]{#2}\n", tex_file);
   fputs("\\newcommand{\\NWlink}[2]{#2}\n", tex_file);
}
fputs("\\newcommand{\\NWtxtMacroDefBy}{Fragment defined by}\n", tex_file);
fputs("\\newcommand{\\NWtxtMacroRefIn}{Fragment referenced in}\n", tex_file);
fputs("\\newcommand{\\NWtxtMacroNoRef}{Fragment never referenced}\n", tex_file);
fputs("\\newcommand{\\NWtxtDefBy}{Defined by}\n", tex_file);
fputs("\\newcommand{\\NWtxtRefIn}{Referenced in}\n", tex_file);
fputs("\\newcommand{\\NWtxtNoRef}{Not referenced}\n", tex_file);
fputs("\\newcommand{\\NWtxtFileDefBy}{File defined by}\n", tex_file);
fputs("\\newcommand{\\NWtxtIdentsUsed}{Uses:}\n", tex_file);
fputs("\\newcommand{\\NWtxtIdentsNotUsed}{Never used}\n", tex_file);
fputs("\\newcommand{\\NWtxtIdentsDefed}{Defines:}\n", tex_file);
fputs("\\newcommand{\\NWsep}{${\\diamond}$}\n", tex_file);
fputs("\\newcommand{\\NWnotglobal}{(not defined globally)}\n", tex_file);
fputs("\\newcommand{\\NWuseHyperlinks}{", tex_file);
if (hyperoptions[0] != '\0')
{
   ⟨ Write the hyperlink usage macro 41b ⟩
}
fputs("}\n", tex_file);
◇
```

Fragment referenced in 40b, 70a.
Uses: `fputs` 11, `hyperoptions` 16, `hyperref_flag` 16, `Uses` 144c.

⟨ *Write the hyperlink usage macro* 41b ⟩ ≡

```
fprintf(tex_file, "\\usepackage[%s]{hyperref}", hyperoptions);◇
```

Fragment referenced in 41a.
Uses: `fprintf` 11, `hyperoptions` 16.

We make our second (and final) pass through the source web, this time copying characters straight into the `.tex` file. However, we keep an eye peeled for `@` characters, which signal a command sequence.

⟨ *Copy* `source_file` *into* `tex_file` 41c ⟩ ≡

```
{
  int inBlock = FALSE;
  int c = source_get();
  while (c != EOF) {
    if (c == nw_char)
      {
      ⟨ Interpret at-sequence 42a ⟩
      }
    else {
      putc(c, tex_file);
      c = source_get();
    }
  }
}◇
```

Fragment referenced in 40b.
Uses: `FALSE` 12a, `nw_char` 17bc, `putc` 11, `source_get` 87c.

⟨ *Interpret at-sequence* 42a ⟩ ≡

```
{
  int big_definition = FALSE;
  c = source_get();
  switch (c) {
    case 'r':
          c = source_get();
          nw_char = c;
          update_delimit_scrap();
          break;
    case 'O': big_definition = TRUE;
    case 'o': ⟨ Write output file definition 44a ⟩
              break;
    case 'Q':
    case 'D': big_definition = TRUE;
    case 'q':
    case 'd': ⟨ Write macro definition 44b ⟩
              break;
    case 's':
              ⟨ Step to next sector 27a ⟩
              break;
    case 'S':
              ⟨ Close the current sector 27b ⟩
              break;
    case '{':
    case '[':
    case '(': ⟨ Write in-text scrap 48c ⟩
              break;
    case 'x': ⟨ Copy label from source into (42b tex_file ) 57b ⟩
              c = source_get();
              break;
    case 'c': ⟨ Begin or end a block comment 34c ⟩
              c = source_get();
              break;
    case 'f': ⟨ Write index of file names 62a ⟩
              break;
    case 'm': ⟨ Write index of macro names 63b ⟩
              break;
    case 'u': ⟨ Write index of user-specified names 67b ⟩
              break;
    case 'v': ⟨ Copy version info into tex file 43 ⟩
              break;
    default:
          if (c==nw_char)
            putc(c, tex_file);
          c = source_get();
              break;
  }
}◇
```

Fragment referenced in 41c.

⟨ *Copy version info into tex file* 43 ⟩ ≡

```
fputs(version_string, tex_file);
c = source_get();
◇
```

Fragment referenced in 42a.
Uses: `fputs` 11, `source_get` 87c, `version_string` 16.

2.4.1 Formatting Definitions

We go through a fair amount of effort to format a file definition. I've derived most of the LaTeX commands experimentally; it's quite likely that an expert could do a better job. The LaTeX for the previous fragment definition should look like this (perhaps modulo the scrap references):

```
\begin{flushleft} \small
\begin{minipage}{\linewidth}\label{scrap70}\raggedright\small
$\langle$Interpret at-sequence {\footnotesize 18}$\rangle\equiv$
\vspace{-1ex}
\begin{list}{}{} \item
\mbox{}\verb@{@\\
\mbox{}\verb@  int big_definition = FALSE;@\\
\mbox{}\verb@  c = source_get();@\\
\mbox{}\verb@  switch (c) {@\\
\mbox{}\verb@    case 'O': big_definition = TRUE;@\\
\mbox{}\verb@    case 'o': @$\langle$Write output file definition {\footnotesize 19a}$\rangle$\verb@@\\

⋮

\mbox{}\verb@    case '@{\tt @}\verb@': putc(c, tex_file);@\\
\mbox{}\verb@    default:  c = source_get();@\\
\mbox{}\verb@              break;@\\
\mbox{}\verb@  }@\\
\mbox{}\verb@}@$\diamond$
\end{list}
\vspace{-1ex}
\footnotesize\addtolength{\baselineskip}{-1ex}
\begin{list}{}{\setlength{\itemsep}{-\parsep}\setlength{\itemindent}{-\leftmargin}}
\item Fragment referenced in scrap 17b.
\end{list}
\end{minipage}\vspace{4ex}
\end{flushleft}
```

The *flushleft* environment is used to avoid LaTeX warnings about underful lines. The *minipage* environment is used to avoid page breaks in the middle of scraps. The *verb* command allows arbitrary characters to be printed (however, note the special handling of the `@` case in the switch statement).

Fragment and file definitions are formatted nearly identically. I've factored the common parts out into separate scraps.

⟨ *Write output file definition* 44a ⟩ ≡

```
{
  Name *name = collect_file_name();
  ⟨ Begin the scrap environment 46a ⟩
  fputs("\\NWtarget{nuweb", tex_file);
  write_single_scrap_ref(tex_file, scraps);
  fputs("}{} ", tex_file);
  fprintf(tex_file, "\\verb%c\"%s\"%c\\nobreak\\ {\\footnotesize {", nw_char, name->spelling, nw_char);
  write_single_scrap_ref(tex_file, scraps);
  fputs("}}$\\equiv$\n", tex_file);
  ⟨ Fill in the middle of the scrap environment 47 ⟩
  ⟨ Begin the cross-reference environment 48d ⟩
  if ( scrap_flag ) {
    ⟨ Write file defs 49b ⟩
  }
  format_defs_refs(tex_file, scraps);
  format_uses_refs(tex_file, scraps++);
  ⟨ Finish the cross-reference environment 49a ⟩
  ⟨ Finish the scrap environment 48a ⟩
}◇
```

Fragment referenced in 42a.
Uses: `collect_file_name` 124, `format_defs_refs` 146b, `format_uses_refs` 145a, `fprintf` 11, `fputs` 11, `Name` 117a, `nw_char` 17bc, `scraps` 93a, `scrap_flag` 16, `write_single_scrap_ref` 94b.

I don't format a fragment name at all specially, figuring the programmer might want to use italics or bold face in the midst of the name.

⟨ *Write macro definition* 44b ⟩ ≡

```
{
  Name *name = collect_macro_name();

  ⟨ Begin the scrap environment 46a ⟩
  fputs("\\NWtarget{nuweb", tex_file);
  write_single_scrap_ref(tex_file, scraps);
  fputs("}{} $\\langle\\,${\\it ", tex_file);
  ⟨ Write the macro's name 45a ⟩
  fputs("}\\nobreak\\ {\\footnotesize {", tex_file);
  write_single_scrap_ref(tex_file, scraps);
  fputs("}}$\\,\\rangle\\equiv$\n", tex_file);
  ⟨ Fill in the middle of the scrap environment 47 ⟩
  ⟨ Begin the cross-reference environment 48d ⟩
  ⟨ Write macro defs 49c ⟩
  ⟨ Write macro refs 50a ⟩
  format_defs_refs(tex_file, scraps);
  format_uses_refs(tex_file, scraps++);
  ⟨ Finish the cross-reference environment 49a ⟩
  ⟨ Finish the scrap environment 48a ⟩
}◇
```

Fragment referenced in 42a.
Uses: `collect_macro_name` 126, `format_defs_refs` 146b, `format_uses_refs` 145a, `fputs` 11, `Name` 117a, `scraps` 93a, `write_single_scrap_ref` 94b.

⟨ *Write the macro's name* 45a ⟩ ≡

```
{
  char * p = name->spelling;
  int i = 0;

  while (*p != '\000') {
    if (*p == ARG_CHR) {
      write_arg(tex_file, name->arg[i++]);
      p++;
    }
    else
       fputc(*p++, tex_file);
  }
}◇
```

Fragment referenced in 44b, 65b.
Uses: `ARG_CHR` 127c, `write_arg` 45b.

"latex.c" 45b≡

```
static void write_arg(FILE * tex_file, char * p)
{
   fputs("\\hbox{\\slshape\\sffamily ", tex_file);
   while (*p)
   {
      switch (*p)
      {
      case '$':
      case '_':
      case '^':
      case '#':
         fputc('\\', tex_file);
         break;
      default:
         break;
      }
      fputc(*p, tex_file);
      p++;
   }

   fputs("\\/}", tex_file);
}
◇
```

File defined by 12e, 40ab, 45b, 50b, 51, 52ac, 53, 54, 55a, 61, 62b, 64ab, 67ac.
Defines: `write_arg` 40a, 45a, 56a, 61.
Uses: `FILE` 11, `fputs` 11.

⟨ *Begin the scrap environment* 46a ⟩ ≡

```
{
  if (big_definition)
  {
    if (inBlock)
    {
       ⟨ End block 46d ⟩
    }
    fputs("\\begin{flushleft} \\small", tex_file);
  }
  else
  {
    if (inBlock)
    {
       ⟨ Switch block 46c ⟩
    }
    else
    {
       ⟨ Start block 46b ⟩
    }
  }
  fprintf(tex_file, "\\label{scrap%d}\\raggedright\\small\n", scraps);
}◇
```

Fragment referenced in 44ab.
Uses: `fprintf` 11, `fputs` 11, `scraps` 93a.

⟨ *Start block* 46b ⟩ ≡

```
fputs("\\begin{flushleft} \\small\n\\begin{minipage}{\\linewidth}", tex_file);
inBlock = TRUE;◇
```

Fragment referenced in 34c, 46a.
Uses: `fputs` 11, `TRUE` 12a.

⟨ *Switch block* 46c ⟩ ≡

```
fputs("\\par\\vspace{\\baselineskip}\n",  tex_file);◇
```

Fragment referenced in 46a.
Uses: `fputs` 11.

⟨ *End block* 46d ⟩ ≡

```
fputs("\\end{minipage}\\vspace{4ex}\n",  tex_file);
fputs("\\end{flushleft}\n", tex_file);
inBlock = FALSE;◇
```

Fragment referenced in 34c, 46a, 48a.
Uses: `FALSE` 12a, `fputs` 11.

The interesting things here are the ◇ inserted at the end of each scrap and the various spacing commands. The diamond helps to clearly indicate the end of a scrap. The spacing commands were derived empirically; they may be adjusted to taste.

⟨ *Fill in the middle of the scrap environment* 47 ⟩ ≡

```
{
  fputs("\\vspace{-1ex}\n\\begin{list}{}{} \\item\n", tex_file);
  extra_scraps = 0;
  copy_scrap(tex_file, TRUE, name);
  fputs("{\\NWsep}\n\\end{list}\n", tex_file);
}◇
```

Fragment referenced in 44ab.
Uses: `copy_scrap` 54, 76, `fputs` 11, `TRUE` 12a.

We've got one last spacing command, controlling the amount of white space after a scrap.

Note also the whitespace eater. I use it to remove any blank lines that appear after a scrap in the source file. This way, text following a scrap will not be indented. Again, this is a matter of personal taste.

⟨ *Finish the scrap environment* 48a ⟩ ≡

```
{
  scraps += extra_scraps;
  if (big_definition)
    fputs("\\vspace{4ex}\n\\end{flushleft}\n", tex_file);
  else
  {
     ⟨ End block 46d ⟩
  }
  do
    c = source_get();
  while (isspace(c));
}◇
```

Fragment referenced in 44ab.
Uses: `fputs` 11, `isspace` 11, `scraps` 93a, `source_get` 87c.

⟨ *Global variable declarations* 48b ⟩ ≡

```
extern int extra_scraps;
◇
```

Fragment defined by 16, 17bd, 27c, 32c, 48b, 86b, 95a, 117b, 151a.
Fragment referenced in 10.

⟨ *Write in-text scrap* 48c ⟩ ≡

```
copy_scrap(tex_file, FALSE, NULL);
c = source_get();
◇
```

Fragment referenced in 42a.
Uses: `copy_scrap` 54, 76, `FALSE` 12a, `source_get` 87c.

Formatting Cross References

⟨ *Begin the cross-reference environment* 48d ⟩ ≡

```
{
  fputs("\\vspace{-1.5ex}\n", tex_file);
  fputs("\\footnotesize\n", tex_file);
  fputs("\\begin{list}{}{\\setlength{\\itemsep}{-\\parsep}",
    tex_file);
  fputs("\\setlength{\\itemindent}{-\\leftmargin}}\n", tex_file);}
◇
```

Fragment referenced in 44ab.
Uses: `fputs` 11.

There may (unusually) be nothing to output in the cross-reference section, so output an empty list item anyway to avoid having an empty list.

⟨ *Finish the cross-reference environment* 49a ⟩ ≡

```
{
  fputs("\n\\item{}", tex_file);
  fputs("\n\\end{list}\n", tex_file);
}◇
```

Fragment referenced in 44ab.
Uses: `fputs` 11.

⟨ *Write file defs* 49b ⟩ ≡

```
{
  if (name->defs) {
    if (name->defs->next) {
      fputs("\\item \\NWtxtFileDefBy\\ ", tex_file);
      print_scrap_numbers(tex_file, name->defs);
    }
  } else {
    fprintf(stderr,
            "would have crashed in 'Write file defs' for '%s'\n",
             name->spelling);
  }
}◇
```

Fragment referenced in 44a.
Uses: `fprintf` 11, `fputs` 11, `print_scrap_numbers` 50b, 75c, `stderr` 11.

⟨ *Write macro defs* 49c ⟩ ≡

```
{
  if (name->defs->next) {
    fputs("\\item \\NWtxtMacroDefBy\\ ", tex_file);
    print_scrap_numbers(tex_file, name->defs);
  }
}◇
```

Fragment referenced in 44b.
Uses: `fputs` 11, `print_scrap_numbers` 50b, 75c.

⟨ *Write macro refs* 50a ⟩ ≡

```
{
  if (name->uses) {
    if (name->uses->next) {
      fputs("\\item \\NWtxtMacroRefIn\\ ", tex_file);
      print_scrap_numbers(tex_file, name->uses);
    }
    else {
      fputs("\\item \\NWtxtMacroRefIn\\ ", tex_file);
      fputs("\\NWlink{nuweb", tex_file);
      write_single_scrap_ref(tex_file, name->uses->scrap);
      fputs("}{", tex_file);
      write_single_scrap_ref(tex_file, name->uses->scrap);
      fputs("}", tex_file);
      fputs(".\n", tex_file);
    }
  }
  else {
    fputs("\\item {\\NWtxtMacroNoRef}.\n", tex_file);
    fprintf(stderr, "%s: <%s> never referenced.\n",
            command_name, name->spelling);
  }
}◇
```

Fragment referenced in 44b.
Uses: `command_name` 17d, `fprintf` 11, `fputs` 11, `print_scrap_numbers` 50b, 75c, `stderr` 11, `write_single_scrap_ref` 94b.

"latex.c" 50b≡

```
static void print_scrap_numbers(tex_file, scraps)
     FILE *tex_file;
     Scrap_Node *scraps;
{
  int page;
  fputs("\\NWlink{nuweb", tex_file);
  write_scrap_ref(tex_file, scraps->scrap, -1, &page);
  fputs("}{", tex_file);
  write_scrap_ref(tex_file, scraps->scrap, TRUE, &page);
  fputs("}", tex_file);
  scraps = scraps->next;
  while (scraps) {
    fputs("\\NWlink{nuweb", tex_file);
    write_scrap_ref(tex_file, scraps->scrap, -1, &page);
    fputs("}{", tex_file);
    write_scrap_ref(tex_file, scraps->scrap, FALSE, &page);
    scraps = scraps->next;
    fputs("}", tex_file);
  }
  fputs(".\n", tex_file);
}
◇
```

File defined by 12e, 40ab, 45b, 50b, 51, 52ac, 53, 54, 55a, 61, 62b, 64ab, 67ac.
Defines: `print_scrap_numbers` 40a, 49bc, 50a, 63a, 66b, 69b, 74abc, 75c, 80bd.
Uses: `FALSE` 12a, `FILE` 11, `fputs` 11, `scraps` 93a, `Scrap_Node` 116d, `TRUE` 12a, `write_scrap_ref` 94a.

Formatting a Scrap

We add a `\mbox{}` at the beginning of each line to avoid problems with older versions of TEX. This is the only place we really care whether a scrap is delimited with `@{...@}`, `@[...@]`, or `@(...@)`, and we base our output sequences on that.

We have an array `delimit_scrap` where we store our strings that perform the formatting for one line of a scrap. Upon initialisation we copy our strings from the source `orig_delimit_scrap` to it to have the strings writeable. We might change the strings later when we encounter the command to change the 'nuweb character'.

"latex.c" 51≡

```
static char *orig_delimit_scrap[3][5] = {
  /* {} mode: begin, end, insert nw_char, prefix, suffix */
  { "\\verb@", "@", "@{\\tt @}\\verb@", "\\mbox{}", "\\\\" },
  /* [] mode: begin, end, insert nw_char, prefix, suffix */
  { "", "", "@", "", "" },
  /* () mode: begin, end, insert nw_char, prefix, suffix */
  { "$", "$", "@", "", "" },
};

static char *delimit_scrap[3][5];
◇
```

File defined by 12e, 40ab, 45b, 50b, 51, 52ac, 53, 54, 55a, 61, 62b, 64ab, 67ac.
Uses: `nw_char` 17bc.

The function `initialise_delimit_scrap_array` does the copying. If we want to have the listings package do the formatting we have to replace only two of those strings: the `verb` command has to be replaced by the package's `lstinline` command.

"latex.c" 52a≡

```
void initialise_delimit_scrap_array() {
  int i,j;
  for(i = 0; i < 3; i++) {
    for(j = 0; j < 5; j++) {
      if((delimit_scrap[i][j] = strdup(orig_delimit_scrap[i][j])) == NULL) {
        fprintf(stderr, "Not enough memory for string allocation\n");
        exit(EXIT_FAILURE);
      }
    }
  }

  /* replace verb by lstinline */
  if (listings_flag) {
    free(delimit_scrap[0][0]);
    if((delimit_scrap[0][0]=strdup("\\lstinline@")) == NULL) {
      fprintf(stderr, "Not enough memory for string allocation\n");
      exit(EXIT_FAILURE);
    }
    free(delimit_scrap[0][2]);
    if((delimit_scrap[0][2]=strdup("@{\\tt @}\\lstinline@")) == NULL) {
      fprintf(stderr, "Not enough memory for string allocation\n");
      exit(EXIT_FAILURE);
    }
  }
}
◇
```

File defined by 12e, 40ab, 45b, 50b, 51, 52ac, 53, 54, 55a, 61, 62b, 64ab, 67ac.
Uses: `exit` 11, `fprintf` 11, `stderr` 11.

⟨ *Function prototypes* 52b ⟩ ≡

```
void initialise_delimit_scrap_array(void);
◇
```

Fragment defined by 25a, 39b, 52b, 55b, 69a, 82b, 86a, 93b, 102b, 114b, 118a, 128d, 139a, 148d, 151b.
Fragment referenced in 10.

"latex.c" 52c≡

```
int scrap_type = 0;
◇
```

File defined by 12e, 40ab, 45b, 50b, 51, 52ac, 53, 54, 55a, 61, 62b, 64ab, 67ac.
Defines: `scrap_type` 32d, 54, 56a, 58ab, 59, 61.

"latex.c" 53≡

```
static void write_literal(FILE * tex_file, char * p, int mode)
{
   fprintf(tex_file, "", p);
   fputs(delimit_scrap[mode][0], tex_file);
   while (*p!= '\000') {
     if (*p == nw_char)
       fputs(delimit_scrap[mode][2], tex_file);
     else
       fputc(*p, tex_file);
     p++;
   }
   fputs(delimit_scrap[mode][1], tex_file);
}
```

◇

File defined by 12e, 40ab, 45b, 50b, 51, 52ac, 53, 54, 55a, 61, 62b, 64ab, 67ac.
Defines: `write_literal` 40a, 59, 61.
Uses: `FILE` 11, `fprintf` 11, `fputs` 11, `nw_char` 17bc.

"latex.c" 54≡

```
static void copy_scrap(file, prefix, name)
     FILE *file;
     int prefix;
     Name * name;
{
  int indent = 0;
  int c;
  char ** params = name->arg;
  if (source_last == '{') scrap_type = 0;
  if (source_last == '[') scrap_type = 1;
  if (source_last == '(') scrap_type = 2;
  c = source_get();
  if (prefix) fputs(delimit_scrap[scrap_type][3], file);
  fputs(delimit_scrap[scrap_type][0], file);
  while (1) {
    switch (c) {
      case '\n': fputs(delimit_scrap[scrap_type][1], file);
                 if (prefix) fputs(delimit_scrap[scrap_type][4], file);
                 fputs("\n", file);
                 if (prefix) fputs(delimit_scrap[scrap_type][3], file);
                 fputs(delimit_scrap[scrap_type][0], file);
                 indent = 0;
                 break;
      case '\t': ⟨ Expand tab into spaces 55c ⟩
                 break;
      default:
         if (c==nw_char)
           {
             ⟨ Check at-sequence for end-of-scrap 56a ⟩
             break;
           }
         putc(c, file);
                 indent++;
                 break;
    }
    c = source_get();
  }
}
◇
```

File defined by 12e, 40ab, 45b, 50b, 51, 52ac, 53, 54, 55a, 61, 62b, 64ab, 67ac.
Defines: `copy_scrap` 38, 39a, 40a, 47, 48c, 69b, 73c, 76, `scrap_type` 32d, 52c, 56a, 58ab, 59, 61.
Uses: `FILE` 11, `fputs` 11, `Name` 117a, `nw_char` 17bc, `putc` 11, `source_get` 87c, `source_last` 87c.

When we encounter the command to change the 'nuweb character' we call this function. It updates the scrap formatting directives accordingly.

"latex.c" 55a≡

```
void update_delimit_scrap()
{
  static int been_here_before = 0;

  /* {}-mode begin */
  if (listings_flag) {
    delimit_scrap[0][0][10] = nw_char;
  } else {
    delimit_scrap[0][0][5] = nw_char;
  }
  /* {}-mode end */
  delimit_scrap[0][1][0] = nw_char;
  /* {}-mode insert nw_char */

  delimit_scrap[0][2][0] = nw_char;
  delimit_scrap[0][2][6] = nw_char;

  if (listings_flag) {
    delimit_scrap[0][2][18] = nw_char;
  } else {
    delimit_scrap[0][2][13] = nw_char;
  }

  /* []-mode insert nw_char */
  delimit_scrap[1][2][0] = nw_char;

  /* ()-mode insert nw_char */
  delimit_scrap[2][2][0] = nw_char;
}
◇
```

File defined by 12e, 40ab, 45b, 50b, 51, 52ac, 53, 54, 55a, 61, 62b, 64ab, 67ac.
Defines: `update_delimit_scrap` 26, 42a, 55b, 71.
Uses: `nw_char` 17bc.

⟨ *Function prototypes* 55b ⟩ ≡

```
void update_delimit_scrap();
◇
```

Fragment defined by 25a, 39b, 52b, 55b, 69a, 82b, 86a, 93b, 102b, 114b, 118a, 128d, 139a, 148d, 151b.
Fragment referenced in 10.
Uses: `update_delimit_scrap` 55a.

⟨ *Expand tab into spaces* 55c ⟩ ≡

```
{
  int delta = 8 - (indent % 8);
  indent += delta;
  while (delta > 0) {
    putc(' ', file);
    delta--;
  }
}◇
```

Fragment referenced in 54, 76, 109b.
Uses: `putc` 11.

⟨ *Check at-sequence for end-of-scrap* 56a ⟩ ≡

```
{
  c = source_get();
  switch (c) {
    case 'c': ⟨ Show presence of a block comment 32d ⟩
              break;
    case 'x': ⟨ Copy label from source into (56b file ) 57b ⟩
              break;
    case 'v': ⟨ Copy version info into file 57a ⟩
              break;
    case '+':
    case '-':
    case '*':
    case '|': ⟨ Skip over index entries 57d ⟩
    case ',':
    case ')':
    case ']':
    case '}': fputs(delimit_scrap[scrap_type][1], file);
              return;
    case '<': ⟨ Format macro name 59 ⟩
              break;
    case '%': ⟨ Skip commented-out code 57e ⟩
              break;
    case '_': ⟨ Bold Keyword 58a ⟩
              break;
    case 't': ⟨ Italic "fragment title" 58b ⟩
              break;
    case 'f': ⟨ Italic "file name" 58b ⟩
              break;
    case '1': case '2': case '3':
    case '4': case '5': case '6':
    case '7': case '8': case '9':
              if (name == NULL
                  || name->arg[c - '1'] == NULL) {
                fputs(delimit_scrap[scrap_type][2], file);
                fputc(c,   file);
              }
              else {
                fputs(delimit_scrap[scrap_type][1], file);
                write_arg(file, name->arg[c - '1']);
                fputs(delimit_scrap[scrap_type][0], file);
              }
              break;
    default:
          if (c==nw_char)
            {
              fputs(delimit_scrap[scrap_type][2], file);
              break;
            }
          /* ignore these since pass1 will have warned about them */
              break;
  }
}◇
```

Fragment referenced in 54.

⟨ *Copy version info into file* 57a ⟩ ≡

```
fputs(version_string, file);
◇
```

Fragment referenced in 56a, 109c.
Uses: `fputs` 11, `version_string` 16.

⟨ *Copy label from source into* 57b ⟩ ≡

```
{
   ⟨ Get label from (57c source_get() ) 148b ⟩
   write_label(label_name, @1);
}◇
```

Fragment referenced in 42a, 56a.
Uses: `source_get` 87c.

There's no need to check for errors here, since we will have already pointed out any during the first pass.

⟨ *Skip over index entries* 57d ⟩ ≡

```
{
  do {
    do
      c = source_get();
    while (c != nw_char);
    c = source_get();
  } while (c != '}' && c != ']' && c != ')' );
}◇
```

Fragment referenced in 56a, 77.
Uses: `nw_char` 17bc, `source_get` 87c.

⟨ *Skip commented-out code* 57e ⟩ ≡

```
{
        do
                c = source_get();
        while (c != '\n');
}◇
```

Fragment referenced in 56a, 77, 98.
Uses: `source_get` 87c.

This scrap helps deal with bold keywords:

⟨ *Bold Keyword* 58a ⟩ ≡

```
{
  fputs(delimit_scrap[scrap_type][1],file);
  fprintf(file, "\\hbox{\\sffamily\\bfseries ");
  c = source_get();
  do {
      fputc(c, file);
      c = source_get();
  } while (c != nw_char);
  c = source_get();
  fprintf(file, "}");
  fputs(delimit_scrap[scrap_type][0], file);
}◇
```

Fragment referenced in 56a.
Uses: `fprintf` 11, `fputs` 11, `nw_char` 17bc, `scrap_type` 52c, 54, `source_get` 87c.

⟨ *Italic "whatever"* 58b ⟩ ≡

```
{
  fputs(delimit_scrap[scrap_type][1],file);
  fprintf(file, "\\hbox{\\sffamily\\slshape whatever}");
  fputs(delimit_scrap[scrap_type][0], file);
}◇
```

Fragment referenced in 56a.
Uses: `fprintf` 11, `fputs` 11, `scrap_type` 52c, 54.

⟨ *Format macro name* 59 ⟩ ≡

```
{
  Arglist *args = collect_scrap_name(-1);
  Name *name = args->name;
  char * p = name->spelling;
  Arglist *q = args->args;
  int narg = 0;

  fputs(delimit_scrap[scrap_type][1],file);
  if (prefix)
    fputs("\\hbox{", file);
  fputs("$\\langle\\,${\\it ", file);
  while (*p != '\000') {
    if (*p == ARG_CHR) {
      if (q == NULL) {
         write_literal(file, name->arg[narg], scrap_type);
      }
      else {
        write_ArglistElement(file, q, params);
        q = q->next;
      }
      p++;
      narg++;
    }
    else
       fputc(*p++, file);
  }
  fputs("}\\nobreak\\ ", file);
  if (scrap_name_has_parameters) {
    ⟨ Format macro parameters 38 ⟩
  }
  fprintf(file, "{\\footnotesize ");
  if (name->defs)
    ⟨ Write abbreviated definition list 60 ⟩
  else {
    putc('?', file);
    fprintf(stderr, "%s: never defined <%s>\n",
            command_name, name->spelling);
  }
  fputs("}$\\,\\rangle$", file);
  if (prefix)
     fputs("}", file);
  fputs(delimit_scrap[scrap_type][0], file);
}◇
```

Fragment referenced in 56a.

Uses: `Arglist` 129b, `ARG_CHR` 127c, `collect_scrap_name` 130, `command_name` 17d, `fprintf` 11, `fputs` 11, `Name` 117a, `putc` 11, `scrap_type` 52c, 54, `stderr` 11, `write_ArglistElement` 61, `write_literal` 53.

⟨ *Write abbreviated definition list* 60 ⟩ ≡

```
{
  Scrap_Node *p = name->defs;
  fputs("\\NWlink{nuweb", file);
  write_single_scrap_ref(file, p->scrap);
  fputs("}{", file);
  write_single_scrap_ref(file, p->scrap);
  fputs("}", file);
  p = p->next;
  if (p)
    fputs(", \\ldots\\ ", file);
}◇
```

Fragment referenced in 59, 61.
Uses: `fputs` 11, `Scrap_Node` 116d, `write_single_scrap_ref` 94b.

"latex.c" 61≡

```
static void
write_ArglistElement(FILE * file, Arglist * args, char ** params)
{
  Name *name = args->name;
  Arglist *q = args->args;

  if (name == NULL) {
    char * p = (char*)q;

    if (p[0] == ARG_CHR) {
       write_arg(file, params[p[1] - '1']);
    } else {
       write_literal(file, (char *)q, 0);
    }
  } else if (name == (Name *)1) {
    Scrap_Node * qq = (Scrap_Node *)q;
    qq->quoted = TRUE;
    fputs(delimit_scrap[scrap_type][0], file);
    write_scraps(file, "", qq,
                 -1, "", 0, 0, 0, 0,
                 NULL, params, 0, "");
    fputs(delimit_scrap[scrap_type][1], file);
    extra_scraps++;
    qq->quoted = FALSE;
  } else {
    char * p = name->spelling;
    fputs("$\\langle\\,${\\it ", file);
      while (*p != '\000') {
      if (*p == ARG_CHR) {
        write_ArglistElement(file, q, params);
        q = q->next;
        p++;
      }
      else
         fputc(*p++, file);
    }
    fputs("}\\nobreak\\ ", file);
    if (scrap_name_has_parameters) {
      int c;

      ⟨ Format macro parameters 38 ⟩
    }
    fprintf(file, "{\\footnotesize ");
    if (name->defs)
      ⟨ Write abbreviated definition list 60 ⟩
    else {
      putc('?', file);
      fprintf(stderr, "%s: never defined <%s>\n",
              command_name, name->spelling);
    }
    fputs("}$\\,\\rangle$", file);
  }
}
◇
```

File defined by 12e, 40ab, 45b, 50b, 51, 52ac, 53, 54, 55a, 61, 62b, 64ab, 67ac.
Defines: `write_ArglistElement` 40a, 59.
Uses: `Arglist` 129b, `ARG_CHR` 127c, `command_name` 17d, `FALSE` 12a, `FILE` 11, `fprintf` 11, `fputs` 11, `Name` 117a, `putc` 11, `Scrap_Node` 116d, `scrap_type` 52c, 54, `stderr` 11, `TRUE` 12a, `write_arg` 45b, `write_literal` 53, `write_scraps` 106a.

2.4.2 Generating the Indices

⟨ *Write index of file names* 62a ⟩ ≡

```
{
  if (file_names) {
    fputs("\n{\\small\\begin{list}{}{\\setlength{\\itemsep}{-\\parsep}",
          tex_file);
    fputs("\\setlength{\\itemindent}{-\\leftmargin}}\n", tex_file);
    format_file_entry(file_names, tex_file);
    fputs("\\end{list}}", tex_file);
  }
  c = source_get();
}◇
```

Fragment referenced in 42a.
Uses: `file_names` 117b, `format_file_entry` 62b, `fputs` 11, `source_get` 87c.

"latex.c" 62b≡

```
static void format_file_entry(name, tex_file)
     Name *name;
     FILE *tex_file;
{
  while (name) {
    format_file_entry(name->llink, tex_file);
    ⟨ Format a file index entry 62c ⟩
    name = name->rlink;
  }
}
◇
```

File defined by 12e, 40ab, 45b, 50b, 51, 52ac, 53, 54, 55a, 61, 62b, 64ab, 67ac.
Defines: `format_file_entry` 40a, 62a, 69b.
Uses: `FILE` 11, `Name` 117a.

⟨ *Format a file index entry* 62c ⟩ ≡

```
fputs("\\item ", tex_file);
fprintf(tex_file, "\\verb%c\"%s\"%c ", nw_char, name->spelling, nw_char);
⟨ Write file's defining scrap numbers 63a ⟩
putc('\n', tex_file);◇
```

Fragment referenced in 62b.
Uses: `fprintf` 11, `fputs` 11, `nw_char` 17bc, `putc` 11.

⟨ *Write file's defining scrap numbers* 63a ⟩ ≡

```
{
  Scrap_Node *p = name->defs;
  fputs("{\\footnotesize {\\NWtxtDefBy}", tex_file);
  if (p->next) {
    /* fputs("s ", tex_file); */
      putc(' ', tex_file);
    print_scrap_numbers(tex_file, p);
  }
  else {
    putc(' ', tex_file);
    fputs("\\NWlink{nuweb", tex_file);
    write_single_scrap_ref(tex_file, p->scrap);
    fputs("}{", tex_file);
    write_single_scrap_ref(tex_file, p->scrap);
    fputs("}", tex_file);
    putc('.', tex_file);
  }
  putc('}', tex_file);
}◇
```

Fragment referenced in 62c.
Uses: `fputs` 11, `print_scrap_numbers` 50b, 75c, `putc` 11, `Scrap_Node` 116d, `write_single_scrap_ref` 94b.

⟨ *Write index of macro names* 63b ⟩ ≡

```
{
  unsigned char sector = current_sector;
  int c = source_get();
  if (c == '+')
     sector = 0;
  else
     source_ungetc(&c);
  if (has_sector(macro_names, sector)) {
    fputs("\n{\\small\\begin{list}{}{\\setlength{\\itemsep}{-\\parsep}",
          tex_file);
    fputs("\\setlength{\\itemindent}{-\\leftmargin}}\n", tex_file);
    format_entry(macro_names, tex_file, sector);
    fputs("\\end{list}}", tex_file);
  } else {
    fputs("None.\n", tex_file);
  }
}
c = source_get();
◇
```

Fragment referenced in 42a.
Uses: `current_sector` 27c, `format_entry` 64b, 79c, `fputs` 11, `has_sector` 67a, `macro_names` 117b, `source_get` 87c, `source_ungetc` 88.

"latex.c" 64a≡

```
static int load_entry(Name * name, Name ** nms, int n)
{
   while (name) {
      n = load_entry(name->llink, nms, n);
      nms[n++] = name;
      name = name->rlink;
   }
   return n;
}
◇
```

File defined by 12e, 40ab, 45b, 50b, 51, 52ac, 53, 54, 55a, 61, 62b, 64ab, 67ac.
Defines: `load_entry` 64b.
Uses: `Name` 117a.

"latex.c" 64b≡

```
static void format_entry(name, tex_file, sector)
     Name *name;
     FILE *tex_file;
     unsigned char sector;
{
  Name ** nms = malloc(num_scraps()*sizeof(Name *));
  int n = load_entry(name, nms, 0);
  int i;

  ⟨ Sort nms of size n for ⟨ Rob's ordering 64c ⟩ 65a ⟩
  for (i = 0; i < n; i++)
  {
     Name * name = nms[i];

     ⟨ Format an index entry 65b ⟩
  }
}
◇
```

File defined by 12e, 40ab, 45b, 50b, 51, 52ac, 53, 54, 55a, 61, 62b, 64ab, 67ac.
Defines: `format_entry` 40a, 63b, 69b, 78c, 79ac.
Uses: `FILE` 11, `load_entry` 64a, `malloc` 11, `Name` 117a, `num_scraps` 93a.

⟨ *Rob's ordering* 64c ⟩ ≡

```
robs_strcmp(ki->spelling, kj->spelling) < 0◇
```

Fragment referenced in 64b.
Uses: `robs_strcmp` 121a.

⟨ *Sort key of size n for ordering* 65a ⟩ ≡

```
int j;
for (j = 1; j < n; j++)
{
   int i = j - 1;
   Name * kj = key[j];

   do
   {
      Name * ki = key[i];

      if (ordering)
         break;
      key[i + 1] = ki;
      i -= 1;
   } while (i >= 0);
   key[i + 1] = kj;
}
◇
```

Fragment referenced in 64b.
Uses: Name 117a.

⟨ *Format an index entry* 65b ⟩ ≡

```
if (name->sector == sector){
  fputs("\\item ", tex_file);
  fputs("$\\langle\\,$", tex_file);
  ⟨ Write the macro's name 45a ⟩
  fputs("\\nobreak\\ {\\footnotesize ", tex_file);
  ⟨ Write defining scrap numbers 66a ⟩
  fputs("}$\\,\\rangle$ ", tex_file);
  ⟨ Write referencing scrap numbers 66b ⟩
  putc('\n', tex_file);
}◇
```

Fragment referenced in 64b.
Uses: fputs 11, putc 11.

⟨ *Write defining scrap numbers* 66a ⟩ ≡

```
{
  Scrap_Node *p = name->defs;
  if (p) {
    int page;
    fputs("\\NWlink{nuweb", tex_file);
    write_scrap_ref(tex_file, p->scrap, -1, &page);
    fputs("}{", tex_file);
    write_scrap_ref(tex_file, p->scrap, TRUE, &page);
    fputs("}", tex_file);
    p = p->next;
    while (p) {
      fputs("\\NWlink{nuweb", tex_file);
      write_scrap_ref(tex_file, p->scrap, -1, &page);
      fputs("}{", tex_file);
      write_scrap_ref(tex_file, p->scrap, FALSE, &page);
      fputs("}", tex_file);
      p = p->next;
    }
  }
  else
    putc('?', tex_file);
}◇
```

Fragment referenced in 65b.
Uses: `FALSE` 12a, `fputs` 11, `putc` 11, `Scrap_Node` 116d, `TRUE` 12a, `write_scrap_ref` 94a.

⟨ *Write referencing scrap numbers* 66b ⟩ ≡

```
{
  Scrap_Node *p = name->uses;
  fputs("{\\footnotesize ", tex_file);
  if (p) {
    fputs("{\\NWtxtRefIn}", tex_file);
    if (p->next) {
      /* fputs("s ", tex_file); */
      putc(' ', tex_file);
      print_scrap_numbers(tex_file, p);
    }
    else {
      putc(' ', tex_file);
      fputs("\\NWlink{nuweb", tex_file);
      write_single_scrap_ref(tex_file, p->scrap);
      fputs("}{", tex_file);
      write_single_scrap_ref(tex_file, p->scrap);
      fputs("}", tex_file);
      putc('.', tex_file);
    }
  }
  else
    fputs("{\\NWtxtNoRef}.", tex_file);
  putc('}', tex_file);
}◇
```

Fragment referenced in 65b.
Uses: `fputs` 11, `print_scrap_numbers` 50b, 75c, `putc` 11, `Scrap_Node` 116d, `write_single_scrap_ref` 94b.

"latex.c" 67a≡

```
int has_sector(name, sector)
Name * name;
unsigned char sector;
{
  while(name) {
    if (name->sector == sector)
       return TRUE;
    if (has_sector(name->llink, sector))
       return TRUE;
     name = name->rlink;
   }
   return FALSE;
}
◇
```

File defined by 12e, 40ab, 45b, 50b, 51, 52ac, 53, 54, 55a, 61, 62b, 64ab, 67ac.
Defines: `has_sector` 63b, 67b.
Uses: `FALSE` 12a, `Name` 117a, `TRUE` 12a.

⟨ *Write index of user-specified names* 67b ⟩ ≡

```
{
    unsigned char sector = current_sector;
    c = source_get();
    if (c == '+') {
       sector = 0;
       c = source_get();
    }
    if (has_sector(user_names, sector)) {
      fputs("\n{\\small\\begin{list}{}{\\setlength{\\itemsep}{-\\parsep}",
            tex_file);
      fputs("\\setlength{\\itemindent}{-\\leftmargin}}\n", tex_file);
      format_user_entry(user_names, tex_file, sector);
      fputs("\\end{list}}", tex_file);
    }
}◇
```

Fragment referenced in 42a.
Uses: `current_sector` 27c, `format_user_entry` 67c, 81b, `fputs` 11, `has_sector` 67a, `source_get` 87c, `user_names` 117b.

"latex.c" 67c≡

```
static void format_user_entry(name, tex_file, sector)
     Name *name;
     FILE *tex_file;
     unsigned char sector;
{
  while (name) {
    format_user_entry(name->llink, tex_file, sector);
    ⟨ Format a user index entry 68 ⟩
    name = name->rlink;
  }
}
◇
```

File defined by 12e, 40ab, 45b, 50b, 51, 52ac, 53, 54, 55a, 61, 62b, 64ab, 67ac.
Defines: `format_user_entry` 40a, 67b, 69b, 81ab.
Uses: `FILE` 11, `Name` 117a.

⟨ *Format a user index entry* 68 ⟩ ≡

```
if (name->sector == sector){
  Scrap_Node *uses = name->uses;
  if ( uses || dangling_flag ) {
    int page;
    Scrap_Node *defs = name->defs;
    fprintf(tex_file, "\\item \\verb%c%s%c: ", nw_char,name->spelling,nw_char);
    if (!uses) {
        fputs("(\\underline{", tex_file);
        fputs("\\NWlink{nuweb", tex_file);
        write_single_scrap_ref(tex_file, defs->scrap);
        fputs("}{", tex_file);
        write_single_scrap_ref(tex_file, defs->scrap);
        fputs("})}", tex_file);
        page = -2;
        defs = defs->next;
    }
    else
      if (!defs || uses->scrap < defs->scrap) {
      fputs("\\NWlink{nuweb", tex_file);
      write_scrap_ref(tex_file, uses->scrap, -1, &page);
      fputs("}{", tex_file);
      write_scrap_ref(tex_file, uses->scrap, TRUE, &page);
      fputs("}", tex_file);
      uses = uses->next;
    }
    else {
      if (defs->scrap == uses->scrap)
        uses = uses->next;
      fputs("\\underline{", tex_file);

      fputs("\\NWlink{nuweb", tex_file);
      write_single_scrap_ref(tex_file, defs->scrap);
      fputs("}{", tex_file);
      write_single_scrap_ref(tex_file, defs->scrap);
      fputs("}}", tex_file);
      page = -2;
      defs = defs->next;
    }
    while (uses || defs) {
      if (uses && (!defs || uses->scrap < defs->scrap)) {
        fputs("\\NWlink{nuweb", tex_file);
        write_scrap_ref(tex_file, uses->scrap, -1, &page);
        fputs("}{", tex_file);
        write_scrap_ref(tex_file, uses->scrap, FALSE, &page);
        fputs("}", tex_file);
        uses = uses->next;
      }
      else {
        if (uses && defs->scrap == uses->scrap)
          uses = uses->next;
        fputs(", \\underline{", tex_file);

        fputs("\\NWlink{nuweb", tex_file);
        write_single_scrap_ref(tex_file, defs->scrap);
        fputs("}{", tex_file);
        write_single_scrap_ref(tex_file, defs->scrap);
        fputs("}", tex_file);
```

```
            putc('}', tex_file);
            page = -2;
            defs = defs->next;
          }
        }
        fputs(".\n", tex_file);
      }
    }◇
```

Fragment referenced in 67c.
Uses: `dangling_flag` 16, `FALSE` 12a, `fprintf` 11, `fputs` 11, `nw_char` 17bc, `putc` 11, `Scrap_Node` 116d, `TRUE` 12a, `write_scrap_ref` 94a, `write_single_scrap_ref` 94b.

2.5 Writing the LaTeX File with HTML Scraps

The HTML generated is patterned closely upon the LaTeX generated in the previous section.[1] When a file name ends in `.hw`, the second pass (invoked via a call to `write_html`) copies most of the text from the source file straight into a `.tex` file. Definitions are formatted slightly and cross-reference information is printed out.

⟨ *Function prototypes* 69a ⟩ ≡

```
extern void write_html();
◇
```

Fragment defined by 25a, 39b, 52b, 55b, 69a, 82b, 86a, 93b, 102b, 114b, 118a, 128d, 139a, 148d, 151b.
Fragment referenced in 10.
Uses: `write_html` 70a.

We need a few local function declarations before we get into the body of `write_html`.

"html.c" 69b≡

```
static void copy_scrap();               /* formats the body of a scrap */
static void display_scrap_ref();        /* formats a scrap reference */
static void display_scrap_numbers();    /* formats a list of scrap numbers */
static void print_scrap_numbers();      /* pluralizes scrap formats list */
static void format_entry();             /* formats an index entry */
static void format_file_entry();        /* formats a file index entry */
static void format_user_entry();
◇
```

File defined by 13a, 69b, 70a, 75abc, 76, 79c, 81b.
Uses: `copy_scrap` 54, 76, `display_scrap_numbers` 75b, `display_scrap_ref` 75a, `format_entry` 64b, 79c, `format_file_entry` 62b, `format_user_entry` 67c, 81b, `print_scrap_numbers` 50b, 75c.

The routine `write_html` takes two file names as parameters: the name of the web source file and the name of the `.tex` output file.

[1] While writing this section, I tried to follow Preston's style as displayed in Section 2.4—J. D. R.

"html.c" 70a≡

```
void write_html(file_name, html_name)
     char *file_name;
     char *html_name;
{
  FILE *html_file = fopen(html_name, "w");
  FILE *tex_file = html_file;
  ⟨ Write LaTeX limbo definitions 41a ⟩
  if (html_file) {
    if (verbose_flag)
      fprintf(stderr, "writing %s\n", html_name);
    source_open(file_name);
    ⟨ Copy source_file into html_file 70b ⟩
    fclose(html_file);
  }
  else
    fprintf(stderr, "%s: can't open %s\n", command_name, html_name);
}
◇
```

File defined by 13a, 69b, 70a, 75abc, 76, 79c, 81b.
Defines: write_html 24b, 69a.
Uses: command_name 17d, fclose 11, FILE 11, fopen 11, fprintf 11, source_open 92a, stderr 11, verbose_flag 16.

We make our second (and final) pass through the source web, this time copying characters straight into the .tex file. However, we keep an eye peeled for @ characters, which signal a command sequence.

⟨ Copy source_file into html_file 70b ⟩ ≡

```
{
  int c = source_get();
  while (c != EOF) {
    if (c == nw_char)
      ⟨ Interpret HTML at-sequence 71 ⟩
    else {
      putc(c, html_file);
      c = source_get();
    }
  }
}◇
```

Fragment referenced in 70a.
Uses: nw_char 17bc, putc 11, source_get 87c.

⟨ *Interpret HTML at-sequence* 71 ⟩ ≡

```
{
  c = source_get();
  switch (c) {
    case 'r':
          c = source_get();
          nw_char = c;
          update_delimit_scrap();
          break;
    case 'O':
    case 'o': ⟨ Write HTML output file definition 72a ⟩
          break;
    case 'Q':
    case 'q':
    case 'D':
    case 'd': ⟨ Write HTML macro definition 72c ⟩
          break;
    case 'f': ⟨ Write HTML index of file names 78c ⟩
          break;
    case 'm': ⟨ Write HTML index of macro names 79a ⟩
          break;
    case 'u': ⟨ Write HTML index of user-specified names 81a ⟩
          break;
    default:
          if (c==nw_char)
            putc(c, html_file);
          c = source_get();
          break;
  }
}◇
```

Fragment referenced in 70b.
Uses: `nw_char` 17bc, `putc` 11, `source_get` 87c, `update_delimit_scrap` 55a.

2.5.1 Formatting Definitions

We go through only a little amount of effort to format a definition. The HTML for the previous fragment definition should look like this (perhaps modulo the scrap references):

```
<pre>
<a name="nuweb68">&lt;Interpret HTML at-sequence 68&gt;</a> =
{
  c = source_get();
  switch (c) {
    case 'O':
    case 'o': &lt;Write HTML output file definition <a href="#nuweb69">69</a>&gt;
          break;
    case 'D':
    case 'd': &lt;Write HTML macro definition <a href="#nuweb71">71</a>&gt;
          break;
    case 'f': &lt;Write HTML index of file names <a href="#nuweb86">86</a>&gt;
          break;
    case 'm': &lt;Write HTML index of macro names <a href="#nuweb87">87</a>&gt;
          break;
    case 'u': &lt;Write HTML index of user-specified names <a href="#nuweb93">93</a>&gt;
          break;
```

```
    default:
         if (c==nw_char)
           putc(c, html_file);
         c = source_get();
              break;
  }
}&lt;&gt;</pre>
Fragment referenced in scrap <a href="#nuweb67">67</a>.
<br>
```

Fragment and file definitions are formatted nearly identically. I've factored the common parts out into separate scraps.

⟨ *Write HTML output file definition* 72a ⟩ ≡

```
{
  Name *name = collect_file_name();
  ⟨ Begin HTML scrap environment 73b ⟩
  ⟨ Write HTML output file declaration 72b ⟩
  scraps++;
  ⟨ Fill in the middle of HTML scrap environment 73c ⟩
  ⟨ Write HTML file defs 74a ⟩
  ⟨ Finish HTML scrap environment 73d ⟩
}◇
```

Fragment referenced in 71.
Uses: `collect_file_name` 124, `Name` 117a, `scraps` 93a.

⟨ *Write HTML output file declaration* 72b ⟩ ≡

```
fputs("<a name=\"nuweb", html_file);
write_single_scrap_ref(html_file, scraps);
fprintf(html_file, "\"><code>\"%s\"</code> ", name->spelling);
write_single_scrap_ref(html_file, scraps);
fputs("</a> =\n", html_file);
◇
```

Fragment referenced in 72a.
Uses: `fprintf` 11, `fputs` 11, `scraps` 93a, `write_single_scrap_ref` 94b.

⟨ *Write HTML macro definition* 72c ⟩ ≡

```
{
  Name *name = collect_macro_name();
  ⟨ Begin HTML scrap environment 73b ⟩
  ⟨ Write HTML macro declaration 73a ⟩
  scraps++;
  ⟨ Fill in the middle of HTML scrap environment 73c ⟩
  ⟨ Write HTML macro defs 74b ⟩
  ⟨ Write HTML macro refs 74c ⟩
  ⟨ Finish HTML scrap environment 73d ⟩
}◇
```

Fragment referenced in 71.
Uses: `collect_macro_name` 126, `Name` 117a, `scraps` 93a.

I don't format a fragment name at all specially, figuring the programmer might want to use italics or bold face in the midst of the name. Note that in this implementation, programmers may only use directives in fragment names that are recognized in preformatted text elements (PRE).

Modification 2001–02–15.: I'm interpreting the fragment name as regular LaTex, so that any formatting can be used in it. To use HTML formatting, the `rawhtml` environment should be used.

⟨ *Write HTML macro declaration* 73a ⟩ ≡
```
fputs("<a name=\"nuweb", html_file);
write_single_scrap_ref(html_file, scraps);
fputs("\">&lt;\\end{rawhtml}", html_file);
fputs(name->spelling, html_file);
fputs("\\begin{rawhtml} ", html_file);
write_single_scrap_ref(html_file, scraps);
fputs("&gt;</a> =\n", html_file);
```
◇

Fragment referenced in 72c.
Uses: `fputs` 11, `scraps` 93a, `write_single_scrap_ref` 94b.

⟨ *Begin HTML scrap environment* 73b ⟩ ≡
```
{
  fputs("\\begin{rawhtml}\n", html_file);
  fputs("<pre>\n", html_file);
}
```
◇

Fragment referenced in 72ac.
Uses: `fputs` 11.

The end of a scrap is marked with the characters `<>`.

⟨ *Fill in the middle of HTML scrap environment* 73c ⟩ ≡
```
{
  copy_scrap(html_file, TRUE);
  fputs("&lt;&gt;</pre>\n", html_file);
}
```
◇

Fragment referenced in 72ac.
Uses: `copy_scrap` 54, 76, `fputs` 11, `TRUE` 12a.

The only task remaining is to get rid of the current at command and end the paragraph.

⟨ *Finish HTML scrap environment* 73d ⟩ ≡
```
{
  fputs("\\end{rawhtml}\n", html_file);
  c = source_get(); /* Get rid of current at command. */
}
```
◇

Fragment referenced in 72ac.
Uses: `fputs` 11, `source_get` 87c.

Formatting Cross References

⟨ *Write HTML file defs* 74a ⟩ ≡

```
{
  if (name->defs->next) {
    fputs("\\end{rawhtml}\\NWtxtFileDefBy\\begin{rawhtml} ", html_file);
    print_scrap_numbers(html_file, name->defs);
    fputs("<br>\n", html_file);
  }
}◇
```

Fragment referenced in 72a.
Uses: `fputs` 11, `print_scrap_numbers` 50b, 75c.

⟨ *Write HTML macro defs* 74b ⟩ ≡

```
{
  if (name->defs->next) {
    fputs("\\end{rawhtml}\\NWtxtMacroDefBy\\begin{rawhtml} ", html_file);
    print_scrap_numbers(html_file, name->defs);
    fputs("<br>\n", html_file);
  }
}◇
```

Fragment referenced in 72c.
Uses: `fputs` 11, `print_scrap_numbers` 50b, 75c.

⟨ *Write HTML macro refs* 74c ⟩ ≡

```
{
  if (name->uses) {
    fputs("\\end{rawhtml}\\NWtxtMacroRefIn\\begin{rawhtml} ", html_file);
    print_scrap_numbers(html_file, name->uses);
  }
  else {
    fputs("\\end{rawhtml}{\\NWtxtMacroNoRef}.\\begin{rawhtml}", html_file);
    fprintf(stderr, "%s: <%s> never referenced.\n",
            command_name, name->spelling);
  }
  fputs("<br>\n", html_file);
}◇
```

Fragment referenced in 72c.
Uses: `command_name` 17d, `fprintf` 11, `fputs` 11, `print_scrap_numbers` 50b, 75c, `stderr` 11.

"html.c" 75a≡
```
static void display_scrap_ref(html_file, num)
     FILE *html_file;
     int num;
{
  fputs("<a href=\"#nuweb", html_file);
  write_single_scrap_ref(html_file, num);
  fputs("\">", html_file);
  write_single_scrap_ref(html_file, num);
  fputs("</a>", html_file);
}
```
◇

File defined by 13a, 69b, 70a, 75abc, 76, 79c, 81b.
Defines: `display_scrap_ref` 69b, 75b, 78b, 82a.
Uses: `FILE` 11, `fputs` 11, `write_single_scrap_ref` 94b.

"html.c" 75b≡
```
static void display_scrap_numbers(html_file, scraps)
     FILE *html_file;
     Scrap_Node *scraps;
{
  display_scrap_ref(html_file, scraps->scrap);
  scraps = scraps->next;
  while (scraps) {
    fputs(", ", html_file);
    display_scrap_ref(html_file, scraps->scrap);
    scraps = scraps->next;
  }
}
```
◇

File defined by 13a, 69b, 70a, 75abc, 76, 79c, 81b.
Defines: `display_scrap_numbers` 69b, 75c, 80c.
Uses: `display_scrap_ref` 75a, `FILE` 11, `fputs` 11, `scraps` 93a, `Scrap_Node` 116d.

"html.c" 75c≡
```
static void print_scrap_numbers(html_file, scraps)
     FILE *html_file;
     Scrap_Node *scraps;
{
  display_scrap_numbers(html_file, scraps);
  fputs(".\n", html_file);
}
```
◇

File defined by 13a, 69b, 70a, 75abc, 76, 79c, 81b.
Defines: `print_scrap_numbers` 40a, 49bc, 50ab, 63a, 66b, 69b, 74abc, 80bd.
Uses: `display_scrap_numbers` 75b, `FILE` 11, `fputs` 11, `scraps` 93a, `Scrap_Node` 116d.

Formatting a Scrap

We must translate HTML special keywords into entities in scraps.

"html.c" 76≡

```
static void copy_scrap(file, prefix)
     FILE *file;
{
  int indent = 0;
  int c = source_get();
  while (1) {
    switch (c) {
      case '<' : fputs("&lt;", file);
                 indent++;
                 break;
      case '>' : fputs("&gt;", file);
                 indent++;
                 break;
      case '&' : fputs("&amp;", file);
                 indent++;
                 break;
      case '\n': fputc(c, file);
                 indent = 0;
                 break;
      case '\t': ⟨ Expand tab into spaces 55c ⟩
                 break;
      default:
         if (c==nw_char)
           {
             ⟨ Check HTML at-sequence for end-of-scrap 77 ⟩
             break;
           }
         putc(c, file);
                 indent++;
                 break;
    }
    c = source_get();
  }
}
```

◇

File defined by 13a, 69b, 70a, 75abc, 76, 79c, 81b.
Defines: `copy_scrap` 38, 39a, 40a, 47, 48c, 54, 69b, 73c.
Uses: `FILE` 11, `fputs` 11, `nw_char` 17bc, `putc` 11, `source_get` 87c.

⟨ *Check HTML at-sequence for end-of-scrap* 77 ⟩ ≡

```
{
  c = source_get();
  switch (c) {
    case '+':
    case '-':
    case '*':
    case '|': ⟨ Skip over index entries 57d ⟩
    case ',':
    case '}':
    case ']':
    case ')': return;
    case '_': ⟨ Write HTML bold tag or end 79b ⟩
              break;
    case '1': case '2': case '3':
    case '4': case '5': case '6':
    case '7': case '8': case '9':
              fputc(nw_char, file);
              fputc(c,   file);
              break;
    case '<': ⟨ Format HTML macro name 78a ⟩
              break;
    case '%': ⟨ Skip commented-out code 57e ⟩
              break;
    default:
         if (c==nw_char)
           {
             fputc(c, file);
             break;
           }
          /* ignore these since pass1 will have warned about them */
              break;
  }
}◇
```

Fragment referenced in 76.
Uses: `nw_char` 17bc, `pass1` 25b, `source_get` 87c.

There's no need to check for errors here, since we will have already pointed out any during the first pass.

⟨ *Format HTML macro name* 78a ⟩ ≡

```
{
  Arglist * args = collect_scrap_name(-1);
  Name *name = args->name;
  fputs("&lt;\\end{rawhtml}", file);
  fputs(name->spelling, file);
  if (scrap_name_has_parameters) {
    ⟨ Format HTML macro parameters 39a ⟩
  }
  fputs("\\begin{rawhtml} ", file);
  if (name->defs)
    ⟨ Write HTML abbreviated definition list 78b ⟩
  else {
    putc('?', file);
    fprintf(stderr, "%s: never defined <%s>\n",
            command_name, name->spelling);
  }
  fputs("&gt;", file);
}◇
```

Fragment referenced in 77.
Uses: `Arglist` 129b, `collect_scrap_name` 130, `command_name` 17d, `fprintf` 11, `fputs` 11, `Name` 117a, `putc` 11, `stderr` 11.

⟨ *Write HTML abbreviated definition list* 78b ⟩ ≡

```
{
  Scrap_Node *p = name->defs;
  display_scrap_ref(file, p->scrap);
  if (p->next)
    fputs(", ... ", file);
}◇
```

Fragment referenced in 78a.
Uses: `display_scrap_ref` 75a, `fputs` 11, `Scrap_Node` 116d.

2.5.2 Generating the Indices

⟨ *Write HTML index of file names* 78c ⟩ ≡

```
{
  if (file_names) {
    fputs("\\begin{rawhtml}\n", html_file);
    fputs("<dl compact>\n", html_file);
    format_entry(file_names, html_file, TRUE);
    fputs("</dl>\n", html_file);
    fputs("\\end{rawhtml}\n", html_file);
  }
  c = source_get();
}◇
```

Fragment referenced in 71.
Uses: `file_names` 117b, `format_entry` 64b, 79c, `fputs` 11, `source_get` 87c, `TRUE` 12a.

⟨ *Write HTML index of macro names* 79a ⟩ ≡

```
{
  if (macro_names) {
    fputs("\\begin{rawhtml}\n", html_file);
    fputs("<dl compact>\n", html_file);
    format_entry(macro_names, html_file, FALSE);
    fputs("</dl>\n", html_file);
    fputs("\\end{rawhtml}\n", html_file);
  }
  c = source_get();
}◇
```

Fragment referenced in 71.
Uses: `FALSE` 12a, `format_entry` 64b, 79c, `fputs` 11, `macro_names` 117b, `source_get` 87c.

⟨ *Write HTML bold tag or end* 79b ⟩ ≡

```
{
     static int toggle;
     toggle = ~toggle;
     if( toggle ) {
        fputs( "<b>", file );
     } else {
        fputs( "</b>", file );
     }
}◇
```

Fragment referenced in 77.
Uses: `fputs` 11.

"html.c" 79c≡

```
static void format_entry(name, html_file, file_flag)
     Name *name;
     FILE *html_file;
     int file_flag;
{
  while (name) {
    format_entry(name->llink, html_file, file_flag);
    ⟨ Format an HTML index entry 80a ⟩
    name = name->rlink;
  }
}
◇
```

File defined by 13a, 69b, 70a, 75abc, 76, 79c, 81b.
Defines: `format_entry` 40a, 63b, 64b, 69b, 78c, 79a.
Uses: `FILE` 11, `Name` 117a.

⟨ *Format an HTML index entry* 80a ⟩ ≡

```
{
  fputs("<dt> ", html_file);
  if (file_flag) {
    fprintf(html_file, "<code>\"%s\"</code>\n<dd> ", name->spelling);
    ⟨ Write HTML file's defining scrap numbers 80b ⟩
  }
  else {
    fputs("&lt;\\end{rawhtml}", html_file);
    fputs(name->spelling, html_file);
    fputs("\\begin{rawhtml} ", html_file);
    ⟨ Write HTML defining scrap numbers 80c ⟩
    fputs("&gt;\n<dd> ", html_file);
    ⟨ Write HTML referencing scrap numbers 80d ⟩
  }
  putc('\n', html_file);
}◇
```

Fragment referenced in 79c.
Uses: fprintf 11, fputs 11, putc 11.

⟨ *Write HTML file's defining scrap numbers* 80b ⟩ ≡

```
{
  fputs("\\end{rawhtml}\\NWtxtDefBy\\begin{rawhtml} ", html_file);
  print_scrap_numbers(html_file, name->defs);
}◇
```

Fragment referenced in 80a.
Uses: fputs 11, print_scrap_numbers 50b, 75c.

⟨ *Write HTML defining scrap numbers* 80c ⟩ ≡

```
{
  if (name->defs)
    display_scrap_numbers(html_file, name->defs);
  else
    putc('?', html_file);
}◇
```

Fragment referenced in 80a.
Uses: display_scrap_numbers 75b, putc 11.

⟨ *Write HTML referencing scrap numbers* 80d ⟩ ≡

```
{
  Scrap_Node *p = name->uses;
  if (p) {
    fputs("\\end{rawhtml}\\NWtxtRefIn\\begin{rawhtml} ", html_file);
    print_scrap_numbers(html_file, p);
  }
  else
    fputs("\\end{rawhtml}{\\NWtxtNoRef}.\\begin{rawhtml}", html_file);
}◇
```

Fragment referenced in 80a.
Uses: fputs 11, print_scrap_numbers 50b, 75c, Scrap_Node 116d.

⟨ *Write HTML index of user-specified names* 81a ⟩ ≡

```
{
  if (user_names) {
    fputs("\\begin{rawhtml}\n", html_file);
    fputs("<dl compact>\n", html_file);
    format_user_entry(user_names, html_file, 0/* Dummy */);
    fputs("</dl>\n", html_file);
    fputs("\\end{rawhtml}\n", html_file);
  }
  c = source_get();
}◇
```

Fragment referenced in 71.
Uses: `format_user_entry` 67c, 81b, `fputs` 11, `source_get` 87c, `user_names` 117b.

"html.c" 81b≡

```
static void format_user_entry(name, html_file, sector)
     Name *name;
     FILE *html_file;
{
  while (name) {
    format_user_entry(name->llink, html_file, sector);
    ⟨ Format a user HTML index entry 82a ⟩
    name = name->rlink;
  }
}
◇
```

File defined by 13a, 69b, 70a, 75abc, 76, 79c, 81b.
Defines: `format_user_entry` 40a, 67bc, 69b, 81a.
Uses: `FILE` 11, `Name` 117a.

⟨ *Format a user HTML index entry* 82a ⟩ ≡

```
{
  Scrap_Node *uses = name->uses;
  if (uses) {
    Scrap_Node *defs = name->defs;
    fprintf(html_file, "<dt><code>%s</code>:\n<dd> ", name->spelling);
    if (uses->scrap < defs->scrap) {
      display_scrap_ref(html_file, uses->scrap);
      uses = uses->next;
    }
    else {
      if (defs->scrap == uses->scrap)
        uses = uses->next;
      fputs("<strong>", html_file);
      display_scrap_ref(html_file, defs->scrap);
      fputs("</strong>", html_file);
      defs = defs->next;
    }
    while (uses || defs) {
      fputs(", ", html_file);
      if (uses && (!defs || uses->scrap < defs->scrap)) {
        display_scrap_ref(html_file, uses->scrap);
        uses = uses->next;
      }
      else {
        if (uses && defs->scrap == uses->scrap)
          uses = uses->next;
        fputs("<strong>", html_file);
        display_scrap_ref(html_file, defs->scrap);
        fputs("</strong>", html_file);
        defs = defs->next;
      }
    }
    fputs(".\n", html_file);
  }
}◇
```

Fragment referenced in 81b.
Uses: `display_scrap_ref` 75a, `fprintf` 11, `fputs` 11, `Scrap_Node` 116d.

2.6 Writing the Output Files

⟨ *Function prototypes* 82b ⟩ ≡

```
extern void write_files();
◇
```

Fragment defined by 25a, 39b, 52b, 55b, 69a, 82b, 86a, 93b, 102b, 114b, 118a, 128d, 139a, 148d, 151b.
Fragment referenced in 10.
Uses: `write_files` 83a.

"output.c" 83a≡

```
void write_files(files)
     Name *files;
{
  while (files) {
    write_files(files->llink);
    ⟨ Write out files->spelling 83c ⟩
    files = files->rlink;
  }
}
◇
```

File defined by 13b, 83a.
Defines: write_files 24b, 82b.
Uses: Name 117a.

MAX_INDENT defines the maximum number of leading whitespace characters. This is only a problem when outputting very long lines, possibly by multiple definitions of the same fragment (as in a list of elements which is added to every time a new element is defined).

⟨ Type declarations 83b ⟩ ≡

```
#define MAX_INDENT 500
◇
```

Fragment defined by 12a, 83b, 116d, 117a, 127c, 129b, 132d, 144c, 150c.
Fragment referenced in 10.
Defines: MAX_INDENT 83c, 108a.

⟨ Write out files->spelling 83c ⟩ ≡

```
{
  static char temp_name[FILENAME_MAX];
  static char real_name[FILENAME_MAX];
  static int temp_name_count = 0;
  char indent_chars[MAX_INDENT];
  int temp_file_fd;
  FILE *temp_file;

  ⟨ Find a free temporary file 84a ⟩

  sprintf(real_name, "%s%s%s", dirpath, path_sep, files->spelling);
  if (verbose_flag)
    fprintf(stderr, "writing %s [%s]\n", files->spelling, temp_name);
  write_scraps(temp_file, files->spelling, files->defs, 0, indent_chars,
               files->debug_flag, files->tab_flag, files->indent_flag,
               files->comment_flag, NULL, NULL, 0, files->spelling);
  fclose(temp_file);

  ⟨ Move the temporary file to the target, if required 84b ⟩
}◇
```

Fragment referenced in 83a.
Uses: fclose 11, FILE 11, fprintf 11, MAX_INDENT 83b, stderr 11, verbose_flag 16, write_scraps 106a.

⟨ *Find a free temporary file* 84a ⟩ ≡

```
for( temp_name_count = 0; temp_name_count < 10000; temp_name_count++) {
  sprintf(temp_name,"%s%snw%06d", dirpath, path_sep, temp_name_count);
#ifdef O_EXCL
  if (-1 != (temp_file_fd = open(temp_name, O_CREAT|O_WRONLY|O_EXCL))) {
     temp_file = fdopen(temp_file_fd, "w");
     break;
  }
#else
  if (0 != (temp_file = fopen(temp_name, "a"))) {
     if ( 0L == ftell(temp_file)) {
        break;
     } else {
        fclose(temp_file);
        temp_file = 0;
     }
  }
#endif
}
if (!temp_file) {
  fprintf(stderr, "%s: can't create %s for a temporary file\n",
          command_name, temp_name);
  exit(-1);
}
◇
```

Fragment referenced in 83c.
Uses: `command_name` 17d, `exit` 11, `fclose` 11, `fopen` 11, `fprintf` 11, `stderr` 11.

Note the call to `remove` before `rename` – the ANSI/ISO C standard does *not* guarantee that renaming a file to an existing filename will overwrite the file.

⟨ *Move the temporary file to the target, if required* 84b ⟩ ≡

```
if (compare_flag)
  ⟨ Compare the temp file and the old file 85a ⟩
else {
  remove(real_name);
  ⟨ Rename the temporary file to the target 85b ⟩
}
◇
```

Fragment referenced in 83c.
Uses: `compare_flag` 16, `remove` 11.

Again, we use a call to `remove` before `rename`.

⟨ *Compare the temp file and the old file* 85a ⟩ ≡

```
{
  FILE *old_file = fopen(real_name, "r");
  if (old_file) {
    int x, y;
    temp_file = fopen(temp_name, "r");
    do {
      x = getc(old_file);
      y = getc(temp_file);
    } while (x == y && x != EOF);
    fclose(old_file);
    fclose(temp_file);
    if (x == y)
      remove(temp_name);
    else {
      remove(real_name);
      ⟨ Rename the temporary file to the target 85b ⟩
    }
  }
  else
    ⟨ Rename the temporary file to the target 85b ⟩
}◇
```

Fragment referenced in 84b.
Uses: `fclose` 11, `FILE` 11, `fopen` 11, `getc` 11, `remove` 11.

⟨ *Rename the temporary file to the target* 85b ⟩ ≡

```
if (0 != rename(temp_name, real_name)) {
  fprintf(stderr, "%s: can't rename output file to %s\n",
          command_name, real_name);
}
◇
```

Fragment referenced in 84b, 85a.
Uses: `command_name` 17d, `fprintf` 11, `stderr` 11.

Chapter 3

The Support Routines

3.1 Source Files

3.1.1 Global Declarations

We need two routines to handle reading the source files.

⟨ *Function prototypes* 86a ⟩ ≡
```
extern void source_open(); /* pass in the name of the source file */
extern int source_get();   /* no args; returns the next char or EOF */
extern int source_last;   /* what last source_get() returned. */
extern int source_peek;   /* The next character to get */
```
◇

Fragment defined by 25a, 39b, 52b, 55b, 69a, 82b, 86a, 93b, 102b, 114b, 118a, 128d, 139a, 148d, 151b.
Fragment referenced in 10.
Uses: `source_get` 87c, `source_last` 87c, `source_open` 92a, `source_peek` 87c.

There are also two global variables maintained for use in error messages and such.

⟨ *Global variable declarations* 86b ⟩ ≡
```
extern char *source_name;  /* name of the current file */
extern int source_line;    /* current line in the source file */
```
◇

Fragment defined by 16, 17bd, 27c, 32c, 48b, 86b, 95a, 117b, 151a.
Fragment referenced in 10.
Defines: `source_line` 26, 28c, 29a, 86c, 87c, 88, 89, 90, 91ab, 92a, 96c, 98, 100, 120a, 124, 125ab, 126, 128b, 130, 131,
`source_name` 22, 23, 24ab, 26, 28c, 29a, 86c, 89, 90, 91ab, 92a, 96c, 98, 100, 120a, 124, 125ab, 126, 127a, 128bc, 130, 131.

⟨ *Global variable definitions* 86c ⟩ ≡
```
char *source_name = NULL;
int source_line = 0;
```
◇

Fragment defined by 17ac, 18a, 27d, 86c, 95b, 117c.
Fragment referenced in 14e.
Uses: `source_line` 86b, `source_name` 86b.

3.1.2 Local Declarations

"input.c" 87a≡
```
static FILE *source_file;  /* the current input file */
static int double_at;
static int include_depth;
◇
```
File defined by 14a, 87abc, 88, 92a.
Defines: `double_at` 89, 92a, `include_depth` 90, 91b, 92a, `source_file` 87c, 88, 89, 90, 91ab, 92a.
Uses: `FILE` 11.

"input.c" 87b≡
```
static struct {
  FILE *file;
  char *name;
  int line;
} stack[10];
◇
```
File defined by 14a, 87abc, 88, 92a.
Defines: `stack` 90, 91b.
Uses: `FILE` 11.

3.1.3 Reading a File

The routine `source_get` returns the next character from the current source file. It notices newlines and keeps the line counter `source_line` up to date. It also catches `EOF` and watches for `@` characters. All other characters are immediately returned. We define `source_last` to let us tell which type of scrap we are defining.

"input.c" 87c≡

```
int source_peek;
int source_last;
int source_get()
{
  int c;
  source_last = c = source_peek;
  switch (c) {
    case EOF:  ⟨ Handle EOF 91b ⟩
               return c;
    case '\n': source_line++;
    default:
           if (c==nw_char)
             {
               ⟨ Handle an "at" character 89 ⟩
               return c;
             }
           source_peek = getc(source_file);
               return c;
  }
}
◇
```

File defined by 14a, 87abc, 88, 92a.
Defines: source_get 25c, 26, 27ab, 28c, 29a, 30ab, 31a, 32ab, 36, 38, 39a, 41c, 42a, 43, 48ac, 54, 56a, 57cde, 58a, 62a, 63b, 67b, 70b, 71, 73d, 76, 77, 78c, 79a, 81a, 86a, 90, 91b, 97, 98, 99b, 100, 101c, 124, 125ab, 126, 127ab, 128c, 130, 131, 132a, source_last 38, 39a, 54, 86a, source_peek 30a, 31a, 32a, 86a, 88, 89, 90, 91b, 92a.
Uses: getc 11, nw_char 17bc, source_file 87a, source_line 86b.

source_ungetc pushes a read character back to the source_file.

"input.c" 88≡

```
void source_ungetc(int *c)
{
        ungetc(source_peek, source_file);
        if(*c == '\n')
                source_line--;
   source_peek=*c;
}
◇
```

File defined by 14a, 87abc, 88, 92a.
Defines: source_ungetc 30b, 31a, 32b, 63b.
Uses: source_file 87a, source_line 86b, source_peek 87c.

This whole @ character handling mess is pretty annoying. I want to recognize @i so I can handle include files correctly. At the same time, it makes sense to recognize illegal @ sequences and complain; this avoids ever having to check anywhere else. Unfortunately, I need to avoid tripping over the @@ sequence; hence this whole unsatisfactory double_at business.

⟨ *Handle an "at" character* 89 ⟩ ≡

```
{
  c = getc(source_file);
  if (double_at) {
    source_peek = c;
    double_at = FALSE;
    c = nw_char;
  }
  else
    switch (c) {
      case 'i': ⟨ Open an include file 90 ⟩
                break;
      case '#': case 'f': case 'm': case 'u': case 'v':
      case 'd': case 'o': case 'D': case 'O': case 's':
      case 'q': case 'Q': case 'S': case 't':
      case '+':
      case '-':
      case '*':
      case '\'':
      case '{': case '}': case '<': case '>': case '|':
      case '(': case ')': case '[': case ']':
      case '%': case '_':
      case ':': case ',': case 'x': case 'c':
      case '1': case '2': case '3': case '4': case '5':
      case '6': case '7': case '8': case '9':
      case 'r':
                source_peek = c;
                c = nw_char;
                break;
      default:
            if (c==nw_char)
              {
                source_peek = c;
                double_at = TRUE;
                break;
              }
             fprintf(stderr, "%s: bad %c sequence %c[%d] (%s, line %d)\n",
                     command_name, nw_char, c, c, source_name, source_line);
             exit(-1);
    }
}◇
```

Fragment referenced in 87c.

Uses: `command_name` 17d, `double_at` 87a, `exit` 11, `FALSE` 12a, `fprintf` 11, `getc` 11, `nw_char` 17bc, `source_file` 87a, `source_line` 86b, `source_name` 86b, `source_peek` 87c, `stderr` 11, `TRUE` 12a.

⟨ *Open an include file* 90 ⟩ ≡

```
{
  char name[FILENAME_MAX];
  char fullname[FILENAME_MAX];
  struct incl * p = include_list;

  if (include_depth >= 10) {
    fprintf(stderr, "%s: include nesting too deep (%s, %d)\n",
            command_name, source_name, source_line);
    exit(-1);
  }
  ⟨ Collect include-file name 91a ⟩
  stack[include_depth].file = source_file;
  fullname[0] = '\0';
  for (;;) {
     strcat(fullname, name);
     source_file = fopen(fullname, "r");
     if (source_file || !p)
        break;
     strcpy(fullname, p->name);
     strcat(fullname, "/");
     p = p->next;
  }
  if (!source_file) {
    fprintf(stderr, "%s: can't open include file %s\n",
            command_name, name);
    source_file = stack[include_depth].file;
  }
  else
  {
     stack[include_depth].name = source_name;
     stack[include_depth].line = source_line + 1;
     include_depth++;
     source_line = 1;
     source_name = save_string(fullname);
  }
  source_peek = getc(source_file);
  c = source_get();
}◇
```

Fragment referenced in 89.

Uses: `command_name` 17d, `exit` 11, `fopen` 11, `fprintf` 11, `getc` 11, `incl` 16, `include_depth` 87a, `save_string` 119a, `source_file` 87a, `source_get` 87c, `source_line` 86b, `source_name` 86b, `source_peek` 87c, `stack` 87b, `stderr` 11.

⟨ *Collect include-file name* 91a ⟩ ≡

```
{
    char *p = name;
    do
      c = getc(source_file);
    while (c == ' ' || c == '\t');
    while (isgraph(c)) {
      *p++ = c;
      c = getc(source_file);
    }
    *p = '\0';
    if (c != '\n') {
      fprintf(stderr, "%s: unexpected characters after file name (%s, %d)\n",
              command_name, source_name, source_line);
      exit(-1);
    }
}◇
```

Fragment referenced in 90.
Uses: `command_name` 17d, `exit` 11, `fprintf` 11, `getc` 11, `isgraph` 11, `source_file` 87a, `source_line` 86b, `source_name` 86b, `stderr` 11.

If an `EOF` is discovered, the current file must be closed and input from the next stacked file must be resumed. If no more files are on the stack, the `EOF` is returned.

⟨ *Handle* `EOF` 91b ⟩ ≡

```
{
  fclose(source_file);
  if (include_depth) {
    include_depth--;
    source_file = stack[include_depth].file;
    source_line = stack[include_depth].line;
    source_name = stack[include_depth].name;
    source_peek = getc(source_file);
    c = source_get();
  }
}◇
```

Fragment referenced in 87c.
Uses: `fclose` 11, `getc` 11, `include_depth` 87a, `source_file` 87a, `source_get` 87c, `source_line` 86b, `source_name` 86b, `source_peek` 87c, `stack` 87b.

3.1.4 Opening a File

The routine `source_open` takes a file name and tries to open the file. If unsuccessful, it complains and halts. Otherwise, it sets `source_name`, `source_line`, and `double_at`.

"input.c" 92a≡

```
void source_open(name)
     char *name;
{
  source_file = fopen(name, "r");
  if (!source_file) {
    fprintf(stderr, "%s: couldn't open %s\n", command_name, name);
    exit(-1);
  }
  nw_char = '@';
  source_name = name;
  source_line = 1;
  source_peek = getc(source_file);
  double_at = FALSE;
  include_depth = 0;
}
◇
```

File defined by 14a, 87abc, 88, 92a.
Defines: `source_open` 25b, 40b, 70a, 86a.
Uses: `command_name` 17d, `double_at` 87a, `exit` 11, `FALSE` 12a, `fopen` 11, `fprintf` 11, `getc` 11, `include_depth` 87a, `nw_char` 17bc, `source_file` 87a, `source_line` 86b, `source_name` 86b, `source_peek` 87c, `stderr` 11.

3.2 Scraps

"scraps.c" 92b≡

```
#define SLAB_SIZE 1024

typedef struct slab {
  struct slab *next;
  char chars[SLAB_SIZE];
} Slab;
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `Slab` 92c, 95cd, 96c, 102c, 103a, `SLAB_SIZE` 95d, 102c, 103a, 138c.

"scraps.c" 92c≡

```
typedef struct {
  char *file_name;
  Slab *slab;
  struct uses *uses;
  struct uses *defs;
  int file_line;
  int page;
  char letter;
  unsigned char sector;
} ScrapEntry;
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `ScrapEntry` 93ac, 96c.
Uses: `Slab` 92b.

"scraps.c" 93a≡

```
static ScrapEntry *SCRAP[256];

#define scrap_array(i) SCRAP[(i) >> 8][(i) & 255]

static int scraps;
int num_scraps()
{
   return scraps;
};
⟨ Forward declarations for scraps.c 106b, ... ⟩
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `num_scraps` 64b, 93b, `SCRAP` 93c, 96c, `scraps` 12e, 13a, 16, 38, 39a, 44ab, 46a, 48a, 50b, 72abc, 73a, 75bc, 93c, 96c, 115a, 116ab, 142, 154, `scrap_array` 94a, 96c, 97, 101ab, 107, 108b, 115a, 116ac, 142, 145a, 146b.
Uses: `ScrapEntry` 92c.

⟨ *Function prototypes* 93b ⟩ ≡

```
extern void init_scraps();
extern int collect_scrap();
extern int write_scraps();
extern void write_scrap_ref();
extern void write_single_scrap_ref();
extern int num_scraps();
◇
```

Fragment defined by 25a, 39b, 52b, 55b, 69a, 82b, 86a, 93b, 102b, 114b, 118a, 128d, 139a, 148d, 151b.
Fragment referenced in 10.
Uses: `collect_scrap` 96b, `init_scraps` 93c, `num_scraps` 93a, `write_scraps` 106a, `write_scrap_ref` 94a, `write_single_scrap_ref` 94b.

"scraps.c" 93c≡

```
void init_scraps()
{
  scraps = 1;
  SCRAP[0] = (ScrapEntry *) arena_getmem(256 * sizeof(ScrapEntry));
}
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `init_scraps` 25b, 93b.
Uses: `arena_getmem` 152a, `SCRAP` 93a, `ScrapEntry` 92c, `scraps` 93a.

"scraps.c" 94a≡

```
void write_scrap_ref(file, num, first, page)
     FILE *file;
     int num;
     int first;
     int *page;
{
  if (scrap_array(num).page >= 0) {
    if (first!=0)
      fprintf(file, "%d", scrap_array(num).page);
    else if (scrap_array(num).page != *page)
      fprintf(file, ", %d", scrap_array(num).page);
    if (scrap_array(num).letter > 0)
      fputc(scrap_array(num).letter, file);
  }
  else {
    if (first!=0)
      putc('?', file);
    else
      fputs(", ?", file);
    ⟨ Warn (only once) about needing to rerun after Latex 94c ⟩
  }
  if (first>=0)
  *page = scrap_array(num).page;
}
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `write_scrap_ref` 50b, 66a, 68, 93b, 94b, 146a.
Uses: `FILE` 11, `first` 151d, `fprintf` 11, `fputs` 11, `putc` 11, `scrap_array` 93a.

"scraps.c" 94b≡

```
void write_single_scrap_ref(file, num)
     FILE *file;
     int num;
{
  int page;
  write_scrap_ref(file, num, TRUE, &page);
}
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `write_single_scrap_ref` 38, 44ab, 50a, 60, 63a, 66b, 68, 72b, 73a, 75a, 93b, 112ac, 149a.
Uses: `FILE` 11, `TRUE` 12a, `write_scrap_ref` 94a.

⟨ *Warn (only once) about needing to rerun after Latex* 94c ⟩ ≡

```
{
  if (!already_warned) {
    fprintf(stderr, "%s: you'll need to rerun nuweb after running latex\n",
            command_name);
    already_warned = TRUE;
  }
}◇
```

Fragment referenced in 94a, 115a.
Uses: `already_warned` 95a, `command_name` 17d, `fprintf` 11, `stderr` 11, `TRUE` 12a.

⟨ *Global variable declarations* 95a ⟩ ≡

```
extern int already_warned;
◇
```

Fragment defined by 16, 17bd, 27c, 32c, 48b, 86b, 95a, 117b, 151a.
Fragment referenced in 10.
Defines: `already_warned` 94c, 95b, 115a.

⟨ *Global variable definitions* 95b ⟩ ≡

```
int already_warned = 0;
◇
```

Fragment defined by 17ac, 18a, 27d, 86c, 95b, 117c.
Fragment referenced in 14e.
Uses: `already_warned` 95a.

"scraps.c" 95c≡

```
typedef struct {
  Slab *scrap;
  Slab *prev;
  int index;
} Manager;
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `Manager` 95d, 96ab, 102c, 103a, 105a, 107, 136a, 138a, 142.
Uses: `Slab` 92b.

"scraps.c" 95d≡

```
static void push(c, manager)
     char c;
     Manager *manager;
{
  Slab *scrap = manager->scrap;
  int index = manager->index;
  scrap->chars[index++] = c;
  if (index == SLAB_SIZE) {
    Slab *new = (Slab *) arena_getmem(sizeof(Slab));
    scrap->next = new;
    manager->scrap = new;
    index = 0;
  }
  manager->index = index;
}
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `push` 33a, 36, 96a, 97, 98, 99a, 101cd.
Uses: `arena_getmem` 152a, `Manager` 95c, `Slab` 92b, `SLAB_SIZE` 92b.

"scraps.c" 96a≡
```
static void pushs(s, manager)
     char *s;
     Manager *manager;
{
  while (*s)
    push(*s++, manager);
}
```
◇

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `pushs` 36, 99a, 101d.
Uses: `Manager` 95c, `push` 95d.

"scraps.c" 96b≡
```
int collect_scrap()
{
  int current_scrap, lblseq = 0;
  int depth = 1;
  Manager writer;
  ⟨ Create new scrap, managed by writer 96c ⟩
  ⟨ Accumulate scrap and return scraps++ 97 ⟩
}
```
◇

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `collect_scrap` 27e, 28a, 36, 93b, 132c.
Uses: `Manager` 95c.

⟨ *Create new scrap, managed by* `writer` 96c ⟩ ≡
```
{
  Slab *scrap = (Slab *) arena_getmem(sizeof(Slab));
  if ((scraps & 255) == 0)
    SCRAP[scraps >> 8] = (ScrapEntry *) arena_getmem(256 * sizeof(ScrapEntry));
  scrap_array(scraps).slab = scrap;
  scrap_array(scraps).file_name = save_string(source_name);
  scrap_array(scraps).file_line = source_line;
  scrap_array(scraps).page = -1;
  scrap_array(scraps).letter = 0;
  scrap_array(scraps).uses = NULL;
  scrap_array(scraps).defs = NULL;
  scrap_array(scraps).sector = current_sector;
  writer.scrap = scrap;
  writer.index = 0;
  current_scrap = scraps++;
}
```
◇

Fragment referenced in 96b.
Uses: `arena_getmem` 152a, `current_sector` 27c, `save_string` 119a, `SCRAP` 93a, `ScrapEntry` 92c, `scraps` 93a, `scrap_array` 93a, `Slab` 92b, `source_line` 86b, `source_name` 86b.

⟨ *Accumulate scrap and return* `scraps++` 97 ⟩ ≡

```
{
  int c = source_get();
  while (1) {
    switch (c) {
      case EOF: fprintf(stderr, "%s: unexpect EOF in (%s, %d)\n",
                        command_name, scrap_array(current_scrap).file_name,
                        scrap_array(current_scrap).file_line);
                exit(-1);
      default:
        if (c==nw_char)
          {
            ⟨ Handle at-sign during scrap accumulation 98 ⟩
                break;
          }
        push(c, &writer);
                c = source_get();
                break;
    }
  }
}◇
```

Fragment referenced in 96b.
Uses: `command_name` 17d, `exit` 11, `fprintf` 11, `nw_char` 17bc, `push` 95d, `scrap_array` 93a, `source_get` 87c, `stderr` 11.

⟨ *Handle at-sign during scrap accumulation* 98 ⟩ ≡

```
{
  c = source_get();
  switch (c) {
    case '(':
    case '[':
    case '{': depth++;
              break;
    case '+':
    case '-':
    case '*':
    case '|': ⟨ Collect user-specified index entries 100 ⟩
              /* Fall through */
    case ')':
    case ']':
    case '}': if (--depth > 0)
                break;
              /* else fall through */
    case ',':
              push('\0', &writer);
              scrap_ended_with = c;
              return current_scrap;
    case '<': ⟨ Handle macro invocation in scrap 101c ⟩
              break;
    case '%': ⟨ Skip commented-out code 57e ⟩
              /* emit line break to the output file to keep #line in sync. */
              push('\n', &writer);
              c = source_get();
              break;
    case 'x': ⟨ Get label while collecting scrap 99a ⟩
              break;
    case 'c': ⟨ Include block comment in a scrap 33a ⟩
              break;
    case '1': case '2': case '3':
    case '4': case '5': case '6':
    case '7': case '8': case '9':
    case 'f': case '#': case 'v':
    case 't':
              push(nw_char, &writer);
              break;
    case '_': c = source_get();
              break;
    default :
          if (c==nw_char)
            {
              push(nw_char, &writer);
              push(nw_char, &writer);
              c = source_get();
              break;
            }
          fprintf(stderr, "%s: unexpected %c%c in scrap (%s, %d)\n",
                      command_name, nw_char, c, source_name, source_line);
              exit(-1);
  }
}◇
```

Fragment referenced in 97.
Uses: `command_name` 17d, `exit` 11, `fprintf` 11, `nw_char` 17bc, `push` 95d, `source_get` 87c, `source_line` 86b, `source_name` 86b, `stderr` 11.

⟨ *Get label while collecting scrap* 99a ⟩ ≡

```
{
    ⟨ Get label from (99b source_get() ) 148b ⟩
    ⟨ Save label to label store 149c ⟩
    push(nw_char, &writer);
    push('x', &writer);
    pushs(label_name, &writer);
    push(nw_char, &writer);
}◇
```

Fragment referenced in 98.
Uses: source_get 87c.

⟨ *Collect user-specified index entries* 100 ⟩ ≡

```
{
  do {
    int type = c;
    do {
      char new_name[MAX_NAME_LEN];
      char *p = new_name;
      unsigned int sector = 0;
      do
        c = source_get();
      while (isspace(c));
      if (c != nw_char) {
        Name *name;
        do {
          *p++ = c;
          c = source_get();
        } while (c != nw_char && !isspace(c));
        *p = '\0';
        switch (type) {
        case '*':
           sector = current_sector;
           ⟨ Add user identifier use 101a ⟩
           break;
        case '-':
           ⟨ Add user identifier use 101a ⟩
           /* Fall through */
        case '|':
           sector = current_sector;
           /* Fall through */
        case '+':
           ⟨ Add user identifier definition 101b ⟩
           break;
        }
      }
    } while (c != nw_char);
    c = source_get();
  }while (c == '|' || c == '*' || c == '-' || c == '+');
  if (c != '}' && c != ']' && c != ')') {
    fprintf(stderr, "%s: unexpected %c%c in index entry (%s, %d)\n",
            command_name, nw_char, c, source_name, source_line);
    exit(-1);
  }
}◇
```

Fragment referenced in 98.

Uses: `command_name` 17d, `current_sector` 27c, `exit` 11, `fprintf` 11, `isspace` 11, `MAX_NAME_LEN` 12b, `Name` 117a, `nw_char` 17bc, `source_get` 87c, `source_line` 86b, `source_name` 86b, `stderr` 11.

⟨ *Add user identifier use* 101a ⟩ ≡

```
name = name_add(&user_names, new_name, sector);
if (!name->uses || name->uses->scrap != current_scrap) {
  Scrap_Node *use = (Scrap_Node *) arena_getmem(sizeof(Scrap_Node));
  use->scrap = current_scrap;
  use->next = name->uses;
  name->uses = use;
  add_uses(&(scrap_array(current_scrap).uses), name);
}◇
```

Fragment referenced in 100.
Uses: `add_uses` 144b, `arena_getmem` 152a, `name_add` 122, `scrap_array` 93a, `Scrap_Node` 116d, `user_names` 117b.

⟨ *Add user identifier definition* 101b ⟩ ≡

```
name = name_add(&user_names, new_name, sector);
if (!name->defs || name->defs->scrap != current_scrap) {
  Scrap_Node *def = (Scrap_Node *) arena_getmem(sizeof(Scrap_Node));
  def->scrap = current_scrap;
  def->next = name->defs;
  name->defs = def;
  add_uses(&(scrap_array(current_scrap).defs), name);
}◇
```

Fragment referenced in 100.
Uses: `add_uses` 144b, `arena_getmem` 152a, `name_add` 122, `scrap_array` 93a, `Scrap_Node` 116d, `user_names` 117b.

⟨ *Handle macro invocation in scrap* 101c ⟩ ≡

```
{
  Arglist * args = collect_scrap_name(current_scrap);
  Name *name = args->name;
  ⟨ Save macro name 101d ⟩
  add_to_use(name, current_scrap);
  if (scrap_name_has_parameters) {
    ⟨ Save macro parameters 36 ⟩
  }
  push(nw_char, &writer);
  push('>', &writer);
  c = source_get();
}◇
```

Fragment referenced in 98.
Uses: `add_to_use` 102a, `Arglist` 129b, `collect_scrap_name` 130, `Name` 117a, `nw_char` 17bc, `push` 95d, `source_get` 87c.

⟨ *Save macro name* 101d ⟩ ≡

```
{
  char buff[24];

  push(nw_char, &writer);
  push('<', &writer);
  push(name->sector, &writer);
  sprintf(buff, "%p", args);
  pushs(buff, &writer);
}◇
```

Fragment referenced in 101c.
Uses: `nw_char` 17bc, `push` 95d, `pushs` 96a.

"scraps.c" 102a≡

```
void
add_to_use(Name * name, int current_scrap)
{
  if (!name->uses || name->uses->scrap != current_scrap) {
    Scrap_Node *use = (Scrap_Node *) arena_getmem(sizeof(Scrap_Node));
    use->scrap = current_scrap;
    use->next = name->uses;
    name->uses = use;
  }
}
```

◇

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `add_to_use` 36, 101c, 102b, 133a.
Uses: `arena_getmem` 152a, `Name` 117a, `Scrap_Node` 116d.

⟨ *Function prototypes* 102b ⟩ ≡

```
extern void add_to_use(Name * name, int current_scrap);
```

◇

Fragment defined by 25a, 39b, 52b, 55b, 69a, 82b, 86a, 93b, 102b, 114b, 118a, 128d, 139a, 148d, 151b.
Fragment referenced in 10.
Uses: `add_to_use` 102a, `Name` 117a.

"scraps.c" 102c≡

```
static char pop(manager)
     Manager *manager;
{
  Slab *scrap = manager->scrap;
  int index = manager->index;
  char c = scrap->chars[index++];
  if (index == SLAB_SIZE) {
    manager->prev = scrap;
    manager->scrap = scrap->next;
    index = 0;
  }
  manager->index = index;
  return c;
}
```

◇

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `pop` 33bc, 34b, 37, 105ab, 107, 108c, 109c, 110b, 136c.
Uses: `Manager` 95c, `Slab` 92b, `SLAB_SIZE` 92b.

"scraps.c" 103a≡

```
static void backup(n, manager)
     int n;
     Manager *manager;
{
  Slab *scrap = manager->scrap;
  int index = manager->index;
  if (n > index
      && manager->prev != NULL)
  {
     manager->scrap = manager->prev;
     manager->prev = NULL;
     index += SLAB_SIZE;
  }
  manager->index = (n <= index ? index - n : 0);
}
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `backup` 108c.
Uses: `Manager` 95c, `Slab` 92b, `SLAB_SIZE` 92b.

"scraps.c" 103b≡

```
void
lookup(int n, Arglist * par, char * arg[9], Name **name, Arglist ** args)
{
  int i;
  Arglist * p = par;

  for (i = 0; i < n && p != NULL; i++)
    p = p->next;
  if (p == NULL) {
    char * a = arg[n];

    *name = NULL;
    *args = (Arglist *)a;
  }
  else {
    *name = p->name;
    *args = p->args;
  }
}
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `lookup` 35, 104b.
Uses: `Arglist` 129b, `Name` 117a.

"scraps.c" 104a≡

```
Arglist * instance(Arglist * a, Arglist * par, char * arg[9], int * ch)
{
   if (a != NULL) {
      int changed = 0;
      Arglist *args, *next;
      Name* name;
      ⟨ Set up name, args and next 104b ⟩
      if (changed){
        ⟨ Build a new arglist 104c ⟩
        *ch = 1;
      }
   }

   return a;
}
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: instance 104b, 111.
Uses: Arglist 129b, Name 117a.

⟨ *Set up name, args and next* 104b ⟩ ≡

```
next = instance(a->next, par, arg, &changed);
name = a->name;
if (name == (Name *)1) {
   Embed_Node * q = (Embed_Node *)arena_getmem(sizeof(Embed_Node));
   q->defs = (Scrap_Node *)a->args;
   q->args = par;
   args = (Arglist *)q;
   changed = 1;
} else if (name != NULL)
  args = instance(a->args, par, arg, &changed);
else {
  char * p = (char *)a->args;
   if (p[0] == ARG_CHR) {
      lookup(p[1] - '1', par, arg, &name, &args);
      changed = 1;
   }
   else {
      args = a->args;
   }
}◇
```

Fragment referenced in 104a.
Uses: arena_getmem 152a, Arglist 129b, ARG_CHR 127c, Embed_Node 132d, instance 104a, lookup 103b, Name 117a, Scrap_Node 116d.

⟨ *Build a new arglist* 104c ⟩ ≡

```
a = (Arglist *)arena_getmem(sizeof(Arglist));
a->name = name;
a->args = args;
a->next = next;◇
```

Fragment referenced in 104a.
Uses: arena_getmem 152a, Arglist 129b.

"scraps.c" 105a≡

```
static Arglist *pop_scrap_name(manager, parameters)
     Manager *manager;
     Parameters *parameters;
{
  char name[MAX_NAME_LEN];
  char *p = name;
  int sector = pop(manager);
  int c = pop(manager);
  Arglist * args;

  while (c != nw_char) {
    *p++ = c;
    c = pop(manager);
  }
  *p = '\000';
  if (sscanf(name, "%p", &args) != 1)
  {
    fprintf(stderr,  "%s: found an internal problem (2)\n", command_name);
    exit(-1);
  }
  ⟨ Check for end of scrap name 105b ⟩
  return args;
}
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `pop_scrap_name` 111.
Uses: `Arglist` 129b, `command_name` 17d, `exit` 11, `fprintf` 11, `Manager` 95c, `MAX_NAME_LEN` 12b, `nw_char` 17bc, `Parameters` 34e, `pop` 102c, `stderr` 11.

⟨ *Check for end of scrap name* 105b ⟩ ≡

```
{
  Name *pn;
  c = pop(manager);
  ⟨ Check for macro parameters 37 ⟩
}◇
```

Fragment referenced in 105a.
Uses: `Name` 117a, `pop` 102c.

"scraps.c" 106a≡

```
int write_scraps(file, spelling, defs, global_indent, indent_chars,
                    debug_flag, tab_flag, indent_flag,
                    comment_flag, inArgs, inParams, parameters, title)
     FILE *file;
     char * spelling;
     Scrap_Node *defs;
     int global_indent;
     char *indent_chars;
     char debug_flag;
     char tab_flag;
     char indent_flag;
     unsigned char comment_flag;
     Arglist * inArgs;
     char * inParams[9];
     Parameters parameters;
     char * title;
{
  /* This is in file file name */
  int indent = 0;
  int newline = 1;
  int iter = 0;
  while (defs) {
    ⟨ Copy defs->scrap to file 107 ⟩
    defs = defs->next;
  }
  return indent + global_indent;
}
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: write_scraps 35, 61, 83c, 93b, 111, 113c.
Uses: Arglist 129b, FILE 11, Parameters 34e, Scrap_Node 116d.

⟨ *Forward declarations for scraps.c* 106b ⟩ ≡

```
int delayed_indent = 0;
◇
```

Fragment defined by 106b, 113b, 125c, 143c.
Fragment referenced in 93a.
Defines: delayed_indent 35, 107, 109a, 111, 112ab.

⟨ *Copy* `defs->scrap` *to* `file` 107 ⟩ ≡

```
{
  char c;
  Manager reader;
  Parameters local_parameters = 0;
  int line_number = scrap_array(defs->scrap).file_line;
  ⟨ Insert debugging information if required 108b ⟩
  reader.scrap = scrap_array(defs->scrap).slab;
  reader.index = 0;
  if (delayed_indent)
  {
    ⟨ Insert appropriate indentation 108c ⟩
  }
  c = pop(&reader);
  while (c) {
    switch (c) {
      case '\n':
         if (global_indent >= 0) {
           putc(c, file);
           line_number++;
           newline = 1;
           delayed_indent = 0;
           ⟨ Insert appropriate indentation 108c ⟩
           break;
         } else {
           /* Don't show newlines in embedded fragmants */
           fputs(". . .", file);
           return;
         }
      case '\t': ⟨ Handle tab characters on output 109b ⟩
                 delayed_indent = 0;
                 break;
      default:
         if (c==nw_char)
           {
             ⟨ Check for macro invocation in scrap 109c ⟩
             break;
           }
         putc(c, file);
         if (global_indent >= 0) {
           ⟨ Add more indentation ' ' 108a ⟩
         }
         indent++;
         if (c > ' ') newline = 0;
         delayed_indent = 0;
         break;
    }
    c = pop(&reader);
  }
}◇
```

Fragment referenced in 106a.
Uses: `delayed_indent` 106b, `fputs` 11, `Manager` 95c, `nw_char` 17bc, `Parameters` 34e, `pop` 102c, `putc` 11, `scrap_array` 93a.

We need to make sure that we don't overflow `indent_chars[]`.

⟨ *Add more indentation* **char** 108a ⟩ ≡

```
{
  if (global_indent + indent >= MAX_INDENT) {
    fprintf(stderr,
           "Error! maximum indentation exceeded in \"%s\".\n",
            spelling);
    exit(1);
  }
  indent_chars[global_indent + indent] = char;
}◇
```

Fragment referenced in 107, 109bc.
Uses: `exit` 11, `fprintf` 11, `MAX_INDENT` 83b, `stderr` 11.

⟨ *Insert debugging information if required* 108b ⟩ ≡

```
if (debug_flag) {
  fprintf(file, "\n#line %d \"%s\"\n",
          line_number, scrap_array(defs->scrap).file_name);
  ⟨ Insert appropriate indentation 108c ⟩
}◇
```

Fragment referenced in 107, 109c.
Uses: `fprintf` 11, `scrap_array` 93a.

⟨ *Insert appropriate indentation* 108c ⟩ ≡

```
{
  char c1 = pop(&reader);
  char c2 = pop(&reader);

  if (indent_flag && !(⟨ Indent suppressed 109a ⟩)) {
    ⟨ Put out the indent 108d ⟩
  }
  indent = 0;
  backup(2, &reader);
}◇
```

Fragment referenced in 107, 108b.
Uses: `backup` 103a, `pop` 102c.

⟨ *Put out the indent* 108d ⟩ ≡

```
if (tab_flag)
    for (indent=0; indent<global_indent; indent++)
      putc(' ', file);
  else
    for (indent=0; indent<global_indent; indent++)
      putc(indent_chars[indent], file);
◇
```

Fragment referenced in 35, 108c.
Uses: `putc` 11.

Indent will be suppressed if the next character is a newline or if the next two characters are `@#`. If the next two characters are `@<`, we suppress the indent for now but mark that it may be needed when the next fragment is started.

⟨ *Indent suppressed* 109a ⟩ ≡

```
c1 == '\n'
|| c1 == nw_char && (c2 == '#' || (delayed_indent |= (c2 == '<')))◇
```

Fragment referenced in 108c.
Uses: `delayed_indent` 106b, `nw_char` 17bc.

⟨ *Handle tab characters on output* 109b ⟩ ≡

```
{
  if (tab_flag)
    ⟨ Expand tab into spaces 55c ⟩
  else {
    putc('\t', file);
    if (global_indent >= 0) {
      ⟨ Add more indentation '\t' 108a ⟩
    }
    indent++;
  }
}◇
```

Fragment referenced in 107.
Uses: `putc` 11.

⟨ *Check for macro invocation in scrap* 109c ⟩ ≡

```
{
  c = pop(&reader);
  switch (c) {
    case 't': ⟨ Copy fragment title into file 112c ⟩
              break;
    case 'c': ⟨ Copy block comment from scrap 33b ⟩
              break;
    case 'f': ⟨ Copy file name into file 110c ⟩
              break;
    case 'x': ⟨ Copy label from scrap into file 110a ⟩
    case '_': break;
    case 'v': ⟨ Copy version info into file 57a ⟩
              break;
    case '<': ⟨ Copy macro into file 111 ⟩
              ⟨ Insert debugging information if required 108b ⟩
              break;
    ⟨ Handle macro parameter substitution 35 ⟩
    default:
          if(c==nw_char)
            {
              putc(c, file);
              if (global_indent >= 0) {
                 ⟨ Add more indentation ' ' 108a ⟩
              }
              indent++;
              break;
            }
          /* ignore, since we should already have a warning */
              break;
  }
}◇
```

Fragment referenced in 107.
Uses: `nw_char` 17bc, `pop` 102c, `putc` 11.

⟨ *Copy label from scrap into file* 110a ⟩ ≡

```
{
    ⟨ Get label from (110b pop(&reader) ) 148b ⟩
    write_label(label_name, file);
}◇
```

Fragment referenced in 109c.
Uses: pop 102c.

⟨ *Copy file name into file* 110c ⟩ ≡

```
if (defs->quoted)
    fprintf(file, "%cf", nw_char);
else
    fputs(spelling, file);
◇
```

Fragment referenced in 109c.
Uses: fprintf 11, fputs 11, nw_char 17bc.

⟨ *Copy macro into* `file` 111 ⟩ ≡

```
{
  Arglist *a = pop_scrap_name(&reader, &local_parameters);
  Name *name = a->name;
  int changed;
  Arglist * args = instance(a->args, inArgs, inParams, &changed);
  int i, narg;
  char * p = name->spelling;
  char * * inParams = name->arg;
  Arglist *q = args;

  if (name->mark) {
    fprintf(stderr, "%s: recursive macro discovered involving <%s>\n",
            command_name, name->spelling);
    exit(-1);
  }
  if (name->defs && !defs->quoted) {
    ⟨ Perhaps comment this macro 112a ⟩
    name->mark = TRUE;
    indent = write_scraps(file, spelling, name->defs, global_indent + indent,
                          indent_chars, debug_flag, tab_flag, indent_flag,
                          comment_flag, args, name->arg,
                          local_parameters, name->spelling);
    indent -= global_indent;
    name->mark = FALSE;
  }
  else
  {
    if (delayed_indent)
    {
      for (i = indent + global_indent; --i >= 0; )
         putc(' ', file);
    }

    fprintf(file, "%c<",  nw_char);
    if (name->sector == 0)
       fputc('+', file);
    ⟨ Comment this macro use 113a ⟩
    fprintf(file, "%c>",  nw_char);
    if (!defs->quoted && !tex_flag)
      fprintf(stderr, "%s: macro never defined <%s>\n",
            command_name, name->spelling);
  }
}◇
```

Fragment referenced in 109c.

Uses: `Arglist` 129b, `command_name` 17d, `delayed_indent` 106b, `exit` 11, `FALSE` 12a, `fprintf` 11, `instance` 104a, `Name` 117a, `nw_char` 17bc, `pop_scrap_name` 105a, `putc` 11, `stderr` 11, `tex_flag` 16, `TRUE` 12a, `write_scraps` 106a.

⟨ *Perhaps comment this macro* 112a ⟩ ≡

```
if (comment_flag && newline) {
   ⟨ Perhaps put a delayed indent 112b ⟩
   fputs(comment_begin[comment_flag], file);
   ⟨ Comment this macro use 113a ⟩
   if (xref_flag) {
      putc(' ', file);
      write_single_scrap_ref(file, name->defs->scrap);
   }
   if (comment_end)
      fputs(comment_end[comment_flag], file);
   putc('\n', file);
   if (!delayed_indent)
      for (i = indent + global_indent; --i >= 0; )
         putc(' ', file);
}
◇
```

Fragment referenced in 35, 111.
Uses: `comment_begin` 125c, `comment_end` 125c, `delayed_indent` 106b, `fputs` 11, `putc` 11, `write_single_scrap_ref` 94b, `xref_flag` 16.

⟨ *Perhaps put a delayed indent* 112b ⟩ ≡

```
if (delayed_indent)
   for (i = indent + global_indent; --i >= 0; )
      putc(' ', file);
◇
```

Fragment referenced in 33b, 112a.
Uses: `delayed_indent` 106b, `putc` 11.

⟨ *Copy fragment title into file* 112c ⟩ ≡

```
{
   char * p = title;
   Arglist *q = inArgs;
   int narg;

   ⟨ Comment this macro use 113a ⟩
   if (xref_flag) {
      putc(' ', file);
      write_single_scrap_ref(file, defs->scrap);
   }
}◇
```

Fragment referenced in 109c.
Uses: `Arglist` 129b, `putc` 11, `write_single_scrap_ref` 94b, `xref_flag` 16.

⟨ *Comment this macro use* 113a ⟩ ≡

```
narg = 0;
while (*p != '\000') {
  if (*p == ARG_CHR) {
    if (q == NULL) {
       if (defs->quoted)
          fprintf(file, "%c'%s%c'", nw_char, inParams[narg], nw_char);
       else
          fprintf(file, "'%s'", inParams[narg]);
    }
    else {
      comment_ArglistElement(file, q, defs->quoted);
      q = q->next;
    }
    p++;
    narg++;
  }
  else
     fputc(*p++, file);
}◇
```

Fragment referenced in 111, 112ac.
Uses: `ARG_CHR` 127c, `comment_ArglistElement` 113b, `fprintf` 11, `nw_char` 17bc.

⟨ *Forward declarations for scraps.c* 113b ⟩ ≡

```
static void
comment_ArglistElement(FILE * file, Arglist * args, int quote)
{
  Name *name = args->name;
  Arglist *q = args->args;

  if (name == NULL) {
    if (quote)
       fprintf(file, "%c'%s%c'", nw_char, (char *)q, nw_char);
    else
       fprintf(file, "'%s'", (char *)q);
  } else if (name == (Name *)1) {
     ⟨ Include an embedded scrap in comment 113c ⟩
  } else {
     ⟨ Include a fragment use in comment 114a ⟩
  }
}
◇
```

Fragment defined by 106b, 113b, 125c, 143c.
Fragment referenced in 93a.
Defines: `comment_ArglistElement` 113a, 114a.
Uses: `Arglist` 129b, `FILE` 11, `fprintf` 11, `Name` 117a, `nw_char` 17bc.

⟨ *Include an embedded scrap in comment* 113c ⟩ ≡

```
Embed_Node * e = (Embed_Node *)q;
fputc('{', file);
write_scraps(file, "", e->defs, -1, "", 0, 0, 0, 0, e->args, 0, 1, "");
fputc('}', file);◇
```

Fragment referenced in 113b.
Uses: `Embed_Node` 132d, `write_scraps` 106a.

⟨ *Include a fragment use in comment* 114a ⟩ ≡

```
char * p = name->spelling;
if (quote)
   fputc(nw_char, file);
fputc('<', file);
if (quote && name->sector == 0)
   fputc('+', file);
while (*p != '\000') {
  if (*p == ARG_CHR) {
    comment_ArglistElement(file, q, quote);
    q = q->next;
    p++;
  }
  else
     fputc(*p++, file);
}
if (quote)
   fputc(nw_char, file);
fputc('>', file);◇
```

Fragment referenced in 113b.
Uses: `ARG_CHR` 127c, `comment_ArglistElement` 113b, `nw_char` 17bc.

3.2.1 Collecting Page Numbers

⟨ *Function prototypes* 114b ⟩ ≡

```
extern void collect_numbers();
◇
```

Fragment defined by 25a, 39b, 52b, 55b, 69a, 82b, 86a, 93b, 102b, 114b, 118a, 128d, 139a, 148d, 151b.
Fragment referenced in 10.
Uses: `collect_numbers` 115a.

"scraps.c" 115a≡

```
void collect_numbers(aux_name)
     char *aux_name;
{
  if (number_flag) {
    int i;
    for (i=1; i<scraps; i++)
      scrap_array(i).page = i;
  }
  else {
    FILE *aux_file = fopen(aux_name, "r");
    already_warned = FALSE;
    if (aux_file) {
      char aux_line[500];
      while (fgets(aux_line, 500, aux_file)) {
        int scrap_number;
        int page_number;
        char dummy[50];
        if (3 == sscanf(aux_line, "\\newlabel{scrap%d}{%[^}]}{%d}",
                        &scrap_number, dummy, &page_number)) {
          if (scrap_number < scraps)
            scrap_array(scrap_number).page = page_number;
          else
            ⟨ Warn (only once) about needing to rerun after Latex 94c ⟩
        }
      }
      fclose(aux_file);
      ⟨ Add letters to scraps with duplicate page numbers 115b ⟩
    }
  }
}
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `collect_numbers` 24b, 114b.
Uses: `already_warned` 95a, `FALSE` 12a, `fclose` 11, `FILE` 11, `fopen` 11, `number_flag` 16, `scraps` 93a, `scrap_array` 93a.

⟨ *Add letters to scraps with duplicate page numbers* 115b ⟩ ≡

```
{
   int i = 0;

   ⟨ Step i to the next valid scrap 116a ⟩
   ⟨ For all remaining scraps 116b ⟩ {
      int j = i;
      ⟨ Step j to the next valid scrap 116a ⟩
      ⟨ Perhaps add letters to the page numbers 116c ⟩
      i = j;
   }
}
◇
```

Fragment referenced in 115a.

⟨ *Step i to the next valid scrap* 116a ⟩ ≡

```
do
   i++;
while (i < scraps && scrap_array(i).page == -1);
◇
```

Fragment referenced in 115b.
Uses: `scraps` 93a, `scrap_array` 93a.

⟨ *For all remaining scraps* 116b ⟩ ≡

```
while (i < scraps)◇
```

Fragment referenced in 115b.
Uses: `scraps` 93a.

⟨ *Perhaps add letters to the page numbers* 116c ⟩ ≡

```
if (scrap_array(i).page == scrap_array(j).page) {
   if (scrap_array(i).letter == 0)
      scrap_array(i).letter = 'a';
   scrap_array(j).letter = scrap_array(i).letter + 1;
}
◇
```

Fragment referenced in 115b.
Uses: `scrap_array` 93a.

3.3 Names

⟨ *Type declarations* 116d ⟩ ≡

```
typedef struct scrap_node {
  struct scrap_node *next;
  int scrap;
  char quoted;
} Scrap_Node;
◇
```

Fragment defined by 12a, 83b, 116d, 117a, 127c, 129b, 132d, 144c, 150c.
Fragment referenced in 10.
Defines: `Scrap_Node` 28b, 35, 50b, 60, 61, 63a, 66ab, 68, 75bc, 78b, 80d, 82a, 101ab, 102a, 104b, 106a, 117a, 132cd, 133c, 134a, 142, 144a, 145c, 147a.

⟨ *Type declarations* 117a ⟩ ≡

```
typedef struct name {
  char *spelling;
  struct name *llink;
  struct name *rlink;
  Scrap_Node *defs;
  Scrap_Node *uses;
  char * arg[9];
  int mark;
  char tab_flag;
  char indent_flag;
  char debug_flag;
  unsigned char comment_flag;
  unsigned char sector;
} Name;
```

◇

Fragment defined by 12a, 83b, 116d, 117a, 127c, 129b, 132d, 144c, 150c.
Fragment referenced in 10.
Defines: `Name` 27e, 28a, 35, 44ab, 54, 59, 61, 62b, 64ab, 65a, 67ac, 72ac, 78a, 79c, 81b, 83a, 100, 101c, 102ab, 103b, 104ab, 105b, 111, 113b, 117bc, 118a, 119b, 120b, 122, 123, 124, 126, 128d, 129abc, 131, 132c, 133c, 134b, 137b, 139c, 142, 144bc, 145c, 147a, 148a.
Uses: `Scrap_Node` 116d.

⟨ *Global variable declarations* 117b ⟩ ≡

```
extern Name *file_names;
extern Name *macro_names;
extern Name *user_names;
extern int scrap_name_has_parameters;
extern int scrap_ended_with;
```

◇

Fragment defined by 16, 17bd, 27c, 32c, 48b, 86b, 95a, 117b, 151a.
Fragment referenced in 10.
Defines: `file_names` 24b, 25b, 34d, 62a, 78c, 117c, 124, `macro_names` 25b, 34d, 63b, 79a, 117c, 128b, `user_names` 25b, 34d, 67b, 81a, 101ab, 117c, 139b.
Uses: `Name` 117a.

⟨ *Global variable definitions* 117c ⟩ ≡

```
Name *file_names = NULL;
Name *macro_names = NULL;
Name *user_names = NULL;
int scrap_name_has_parameters;
int scrap_ended_with;
```

◇

Fragment defined by 17ac, 18a, 27d, 86c, 95b, 117c.
Fragment referenced in 14e.
Uses: `file_names` 117b, `macro_names` 117b, `Name` 117a, `user_names` 117b.

⟨ *Function prototypes* 118a ⟩ ≡

```
extern Name *collect_file_name();
extern Name *collect_macro_name();
extern Arglist *collect_scrap_name();
extern Name *name_add();
extern Name *prefix_add();
extern char *save_string();
extern void reverse_lists();
◇
```

Fragment defined by 25a, 39b, 52b, 55b, 69a, 82b, 86a, 93b, 102b, 114b, 118a, 128d, 139a, 148d, 151b.
Fragment referenced in 10.
Uses: `Arglist` 129b, `collect_file_name` 124, `collect_macro_name` 126, `collect_scrap_name` 130, `Name` 117a, `name_add` 122, `prefix_add` 119b, `reverse_lists` 133c, `save_string` 119a.

"names.c" 118b≡

```
enum { LESS, GREATER, EQUAL, PREFIX, EXTENSION };

static int compare(x, y)
     char *x;
     char *y;
{
  int len, result;
  int xl = strlen(x);
  int yl = strlen(y);
  int xp = x[xl - 1] == ' ';
  int yp = y[yl - 1] == ' ';
  if (xp) xl--;
  if (yp) yl--;
  len = xl < yl ? xl : yl;
  result = strncmp(x, y, len);
  if (result < 0) return GREATER;
  else if (result > 0) return LESS;
  else if (xl < yl) {
    if (xp) return EXTENSION;
    else return LESS;
  }
  else if (xl > yl) {
    if (yp) return PREFIX;
    else return GREATER;
  }
  else return EQUAL;
}
◇
```

File defined by 14c, 118b, 119ab, 120b, 121a, 122, 124, 126, 129ac, 130, 133c, 134a.
Defines: `compare` 119b, 120b, `EQUAL` 119b, 120b, `EXTENSION` 119b, 120b, `GREATER` 119b, 120b, `LESS` 119b, 120b, `PREFIX` 119b, 120b.
Uses: `strlen` 11.

"names.c" 119a≡

```
char *save_string(s)
     char *s;
{
  char *new = (char *) arena_getmem((strlen(s) + 1) * sizeof(char));
  strcpy(new, s);
  return new;
}
◇
```

File defined by 14c, 118b, 119ab, 120b, 121a, 122, 124, 126, 129ac, 130, 133c, 134a.
Defines: `save_string` 20b, 24a, 90, 96c, 118a, 119b, 123, 129a, 133b.
Uses: `arena_getmem` 152a, `strlen` 11.

"names.c" 119b≡

```
static int ambiguous_prefix();

static char * found_name = NULL;

Name *prefix_add(rt, spelling, sector)
     Name **rt;
     char *spelling;
     unsigned char sector;
{
  Name *node = *rt;
  int cmp;

  while (node) {
    switch ((cmp = compare(node->spelling, spelling))) {
    case GREATER:   rt = &node->rlink;
                    break;
    case LESS:      rt = &node->llink;
                    break;
    case EQUAL:
                    found_name = node->spelling;
    case EXTENSION: if (node->sector > sector) {
                       rt = &node->rlink;
                       break;
                    }
                    else if (node->sector < sector) {
                       rt = &node->llink;
                       break;
                    }
                    if (cmp == EXTENSION)
                       node->spelling = save_string(spelling);
                    return node;
    case PREFIX:    ⟨ Check for ambiguous prefix 120a ⟩
                    return node;
    }
    node = *rt;
  }
  ⟨ Create new name entry 123 ⟩
}
◇
```

File defined by 14c, 118b, 119ab, 120b, 121a, 122, 124, 126, 129ac, 130, 133c, 134a.
Defines: `prefix_add` 118a, 128b.
Uses: `compare` 118b, `EQUAL` 118b, `EXTENSION` 118b, `GREATER` 118b, `LESS` 118b, `Name` 117a, `PREFIX` 118b, `save_string` 119a.

Since a very short prefix might match more than one fragment name, I need to check for other matches to avoid mistakes. Basically, I simply continue the search down *both* branches of the tree.

⟨ *Check for ambiguous prefix* 120a ⟩ ≡

```
{
  if (ambiguous_prefix(node->llink, spelling, sector) ||
      ambiguous_prefix(node->rlink, spelling, sector))
    fprintf(stderr,
            "%s: ambiguous prefix %c<%s...%c> (%s, line %d)\n",
            command_name, nw_char, spelling, nw_char, source_name, source_line);
}◇
```

Fragment referenced in 119b.
Uses: `command_name` 17d, `fprintf` 11, `nw_char` 17bc, `source_line` 86b, `source_name` 86b, `stderr` 11.

"names.c" 120b≡

```
static int ambiguous_prefix(node, spelling, sector)
     Name *node;
     char *spelling;
     unsigned char sector;
{
  while (node) {
    switch (compare(node->spelling, spelling)) {
    case GREATER:   node = node->rlink;
                    break;
    case LESS:      node = node->llink;
                    break;
    case EXTENSION:
    case PREFIX:
    case EQUAL:     if (node->sector > sector) {
                       node = node->rlink;
                       break;
                    }
                    else if (node->sector < sector) {
                       node = node->llink;
                       break;
                    }
                    return TRUE;
    }
  }
  return FALSE;
}
◇
```

File defined by 14c, 118b, 119ab, 120b, 121a, 122, 124, 126, 129ac, 130, 133c, 134a.
Uses: `compare` 118b, `EQUAL` 118b, `EXTENSION` 118b, `FALSE` 12a, `GREATER` 118b, `LESS` 118b, `Name` 117a, `PREFIX` 118b, `TRUE` 12a.

Rob Shillingsburg suggested that I organize the index of user-specified identifiers more traditionally; that is, not relying on strict ASCII comparisons via `strcmp`. Ideally, we'd like to see the index ordered like this:

aardvark
Adam
atom
Atomic
atoms

The function `robs_strcmp` implements the desired predicate. It returns -2 for alphabetically less-than, -1 for less-than but only differing in case, zero for equal, 1 for greater-than but only in case and 2 for alphabetically greater-than.

"names.c" 121a≡

```
int robs_strcmp(x, y)
     char *x;
     char *y;
{
   int cmp = 0;

   for (; *x && *y; x++, y++)
   {
      ⟨ Skip invisibles on x 121b ⟩
      ⟨ Skip invisibles on y 121b ⟩
      if (*x == *y)
         continue;
      if (islower(*x) && toupper(*x) == *y)
      {
         if (!cmp) cmp = 1;
         continue;
      }
      if (islower(*y) && *x == toupper(*y))
      {
         if (!cmp) cmp = -1;
         continue;
      }
      return 2*(toupper(*x) - toupper(*y));
   }
   if (*x)
      return 2;
   if (*y)
      return -2;
   return cmp;
}
```

◇

File defined by 14c, 118b, 119ab, 120b, 121a, 122, 124, 126, 129ac, 130, 133c, 134a.
Defines: `robs_strcmp` 64c, 122, 123, 144b.
Uses: `islower` 11, `toupper` 11.

Certain character sequences are invisible when printed. We don't want them to be considered for the alphabetical ordering.

⟨ *Skip invisibles on* p 121b ⟩ ≡

```
if (*p == '|')
   p++;
```

◇

Fragment referenced in 121a.

"names.c" 122≡

```
Name *name_add(rt, spelling, sector)
     Name **rt;
     char *spelling;
     unsigned char sector;
{
  Name *node = *rt;
  while (node) {
    int result = robs_strcmp(node->spelling, spelling);
    if (result > 0)
      rt = &node->llink;
    else if (result < 0)
      rt = &node->rlink;
    else
    {
       found_name = node->spelling;
       if (node->sector > sector)
         rt = &node->llink;
       else if (node->sector < sector)
         rt = &node->rlink;
       else
         return node;
    }
    node = *rt;
  }
  ⟨ Create new name entry 123 ⟩
}
◇
```

File defined by 14c, 118b, 119ab, 120b, 121a, 122, 124, 126, 129ac, 130, 133c, 134a.
Defines: `name_add` 101ab, 118a, 124.
Uses: `Name` 117a, `robs_strcmp` 121a.

⟨ *Create new name entry* 123 ⟩ ≡

```
{
  node = (Name *) arena_getmem(sizeof(Name));
  if (found_name && robs_strcmp(found_name, spelling) == 0)
     node->spelling = found_name;
  else
     node->spelling = save_string(spelling);
  node->mark = FALSE;
  node->llink = NULL;
  node->rlink = NULL;
  node->uses = NULL;
  node->defs = NULL;
  node->arg[0] =
  node->arg[1] =
  node->arg[2] =
  node->arg[3] =
  node->arg[4] =
  node->arg[5] =
  node->arg[6] =
  node->arg[7] =
  node->arg[8] = NULL;
  node->tab_flag = TRUE;
  node->indent_flag = TRUE;
  node->debug_flag = FALSE;
  node->comment_flag = 0;
  node->sector = sector;
  *rt = node;
  return node;
}◇
```

Fragment referenced in 119b, 122.
Uses: `arena_getmem` 152a, `FALSE` 12a, `Name` 117a, `robs_strcmp` 121a, `save_string` 119a, `TRUE` 12a.

Name terminated by whitespace. Also check for “per-file” flags. Keep skipping white space until we reach scrap.

"names.c" 124≡

```
Name *collect_file_name()
{
  Name *new_name;
  char name[MAX_NAME_LEN];
  char *p = name;
  int start_line = source_line;
  int c = source_get(), c2;
  while (isspace(c))
    c = source_get();
  while (isgraph(c)) {
    *p++ = c;
    c = source_get();
  }
  if (p == name) {
    fprintf(stderr, "%s: expected file name (%s, %d)\n",
            command_name, source_name, start_line);
    exit(-1);
  }
  *p = '\0';
  /* File names are always global. */
  new_name = name_add(&file_names, name, 0);
  ⟨ Handle optional per-file flags 125a ⟩
  c2 = source_get();
  if (c != nw_char || (c2 != '{' && c2 != '(' && c2 != '[')) {
    fprintf(stderr, "%s: expected %c{, %c[, or %c( after file name (%s, %d)\n",
            command_name, nw_char, nw_char, nw_char, source_name, start_line);
    exit(-1);
  }
  return new_name;
}
◇
```

File defined by 14c, 118b, 119ab, 120b, 121a, 122, 124, 126, 129ac, 130, 133c, 134a.
Defines: `collect_file_name` 27e, 44a, 72a, 118a.
Uses: `command_name` 17d, `exit` 11, `file_names` 117b, `fprintf` 11, `isgraph` 11, `isspace` 11, `MAX_NAME_LEN` 12b, `Name` 117a, `name_add` 122, `nw_char` 17bc, `source_get` 87c, `source_line` 86b, `source_name` 86b, `stderr` 11.

⟨ *Handle optional per-file flags* 125a ⟩ ≡

```
{
  while (1) {
    while (isspace(c))
      c = source_get();
    if (c == '-') {
      c = source_get();
      do {
        switch (c) {
          case 't': new_name->tab_flag = FALSE;
                    break;
          case 'd': new_name->debug_flag = TRUE;
                    break;
          case 'i': new_name->indent_flag = FALSE;
                    break;
          case 'c': ⟨ Get comment delimiters 125b ⟩
                    break;
          default : fprintf(stderr, "%s: unexpected per-file flag (%s, %d)\n",
                            command_name, source_name, source_line);
                    break;
        }
        c = source_get();
      } while (!isspace(c));
    }
    else break;
  }
}◇
```

Fragment referenced in 124.
Uses: command_name 17d, FALSE 12a, fprintf 11, isspace 11, source_get 87c, source_line 86b, source_name 86b, stderr 11, TRUE 12a.

So far we only deal with C comments.

⟨ *Get comment delimiters* 125b ⟩ ≡

```
c = source_get();
if (c == 'c')
   new_name->comment_flag = 1;
else if (c == '+')
   new_name->comment_flag = 2;
else if (c == 'p')
   new_name->comment_flag = 3;
else
   fprintf(stderr, "%s: Unrecognised comment flag (%s, %d)\n",
           command_name, source_name, source_line);
◇
```

Fragment referenced in 125a.
Uses: command_name 17d, fprintf 11, source_get 87c, source_line 86b, source_name 86b, stderr 11.

⟨ *Forward declarations for scraps.c* 125c ⟩ ≡

```
char * comment_begin[4] = { "", "/* ", "// ", "# "};
char * comment_mid[4] = { "", " * ", "// ", "# "};
char * comment_end[4] = { "", " */", "", ""};
◇
```

Fragment defined by 106b, 113b, 125c, 143c.
Fragment referenced in 93a.
Defines: comment_begin 33b, 112a, comment_end 33b, 112a, comment_mid 33b.

Name terminated by `\n` or `@{`; but keep skipping until `@{`

"names.c" 126≡

```
Name *collect_macro_name()
{
  char name[MAX_NAME_LEN];
  char args[1000];
  char * arg[9];
  char * argp = args;
  int argc = 0;
  char *p = name;
  int start_line = source_line;
  int c = source_get(), c2;
  unsigned char sector = current_sector;

  if (c == '+') {
    sector = 0;
    c = source_get();
  }
  while (isspace(c))
    c = source_get();
  while (c != EOF) {
    Name * node;
    switch (c) {
      case '\t':
      case ' ':  *p++ = ' ';
                 do
                   c = source_get();
                 while (c == ' ' || c == '\t');
                 break;
      case '\n': ⟨ Skip until scrap begins, then return name 128c ⟩
      default:
         if (c==nw_char)
           {
             ⟨ Check for terminating at-sequence and return name 127a ⟩
             break;
           }
         *p++ = c;
                 c = source_get();
                 break;
    }
  }
  fprintf(stderr, "%s: expected fragment name (%s, %d)\n",
          command_name, source_name, start_line);
  exit(-1);
  return NULL;  /* unreachable return to avoid warnings on some compilers */
}
◇
```

File defined by 14c, 118b, 119ab, 120b, 121a, 122, 124, 126, 129ac, 130, 133c, 134a.
Defines: `collect_macro_name` 28a, 44b, 72c, 118a.
Uses: `command_name` 17d, `current_sector` 27c, `exit` 11, `fprintf` 11, `isspace` 11, `MAX_NAME_LEN` 12b, `Name` 117a, `nw_char` 17bc, `source_get` 87c, `source_line` 86b, `source_name` 86b, `stderr` 11.

⟨ *Check for terminating at-sequence and return name* 127a ⟩ ≡

```
{
  c = source_get();
  switch (c) {
    case '(':
    case '[':
    case '{': ⟨ Cleanup and install name 128b ⟩
              return install_args(node, argc, arg);
    case '\'': ⟨ Enter the next argument 127b ⟩
               break;
    default:
          if (c==nw_char)
            {
              *p++ = c;
              break;
            }
          fprintf(stderr,
                      "%s: unexpected %c%c in fragment definition name (%s, %d)\n",
                      command_name, nw_char, c, source_name, start_line);
              exit(-1);
  }
}◇
```

Fragment referenced in 126.
Uses: `command_name` 17d, `exit` 11, `fprintf` 11, `nw_char` 17bc, `source_get` 87c, `source_name` 86b, `stderr` 11.

⟨ *Enter the next argument* 127b ⟩ ≡

```
arg[argc] = argp;
while ((c = source_get()) != EOF) {
   if (c==nw_char) {
      c2 = source_get();
      if (c2=='\'') {
        ⟨ Make this argument 128a ⟩
        c = source_get();
        break;
      }
      else
        *argp++ = c2;
   }
   else
     *argp++ = c;
}
*p++ = ARG_CHR;
◇
```

Fragment referenced in 127a.
Uses: `ARG_CHR` 127c, `nw_char` 17bc, `source_get` 87c.

⟨ *Type declarations* 127c ⟩ ≡

```
#define ARG_CHR '\001'
◇
```

Fragment defined by 12a, 83b, 116d, 117a, 127c, 129b, 132d, 144c, 150c.
Fragment referenced in 10.
Defines: `ARG_CHR` 45a, 59, 61, 104b, 113a, 114a, 127b, 131, 132b.

⟨ *Make this argument* 128a ⟩ ≡

```
if (argc < 9) {
  *argp++ = '\000';
  argc += 1;
}
◇
```

Fragment referenced in 127b.

⟨ *Cleanup and install name* 128b ⟩ ≡

```
{
  if (p > name && p[-1] == ' ')
    p--;
  if (p - name > 3 && p[-1] == '.' && p[-2] == '.' && p[-3] == '.') {
    p[-3] = ' ';
    p -= 2;
  }
  if (p == name || name[0] == ' ') {
    fprintf(stderr, "%s: empty name (%s, %d)\n",
            command_name, source_name, source_line);
    exit(-1);
  }
  *p = '\0';
  node = prefix_add(&macro_names, name, sector);
}◇
```

Fragment referenced in 127a, 128c, 131.
Uses: `command_name` 17d, `exit` 11, `fprintf` 11, `macro_names` 117b, `prefix_add` 119b, `source_line` 86b, `source_name` 86b, `stderr` 11.

⟨ *Skip until scrap begins, then return name* 128c ⟩ ≡

```
{
  do
    c = source_get();
  while (isspace(c));
  c2 = source_get();
  if (c != nw_char || (c2 != '{' && c2 != '(' && c2 != '[')) {
    fprintf(stderr, "%s: expected %c{ after fragment name (%s, %d)\n",
            command_name, nw_char, source_name, start_line);
    exit(-1);
  }
  ⟨ Cleanup and install name 128b ⟩
  return install_args(node, argc, arg);
}◇
```

Fragment referenced in 126.
Uses: `command_name` 17d, `exit` 11, `fprintf` 11, `isspace` 11, `nw_char` 17bc, `source_get` 87c, `source_name` 86b, `stderr` 11.

⟨ *Function prototypes* 128d ⟩ ≡

```
extern Name *install_args();
◇
```

Fragment defined by 25a, 39b, 52b, 55b, 69a, 82b, 86a, 93b, 102b, 114b, 118a, 128d, 139a, 148d, 151b.
Fragment referenced in 10.
Uses: `Name` 117a.

"names.c" 129a≡

```
Name *install_args(Name * name, int argc, char *arg[9])
{
  int i;

  for (i = 0; i < argc; i++) {
    if (name->arg[i] == NULL)
      name->arg[i] = save_string(arg[i]);
  }
  return name;
}
◇
```

File defined by 14c, 118b, 119ab, 120b, 121a, 122, 124, 126, 129ac, 130, 133c, 134a.
Uses: Name 117a, save_string 119a.

⟨ *Type declarations* 129b ⟩ ≡

```
typedef struct arglist
{Name * name;
struct arglist * args;
struct arglist * next;
} Arglist;
◇
```

Fragment defined by 12a, 83b, 116d, 117a, 127c, 129b, 132d, 144c, 150c.
Fragment referenced in 10.
Defines: Arglist 35, 59, 61, 78a, 101c, 103b, 104abc, 105a, 106a, 111, 112c, 113b, 118a, 129c, 130, 132cd, 133b, 135d, 136b, 137b, 143b.
Uses: Name 117a.

"names.c" 129c≡

```
Arglist * buildArglist(Name * name, Arglist * a)
{
  Arglist * args = (Arglist *)arena_getmem(sizeof(Arglist));

  args->args = a;
  args->next = NULL;
  args->name = name;
  return args;
}
◇
```

File defined by 14c, 118b, 119ab, 120b, 121a, 122, 124, 126, 129ac, 130, 133c, 134a.
Defines: buildArglist 131, 132c, 133b.
Uses: arena_getmem 152a, Arglist 129b, Name 117a.

Terminated by @>

"names.c" 130≡

```
Arglist * collect_scrap_name(int current_scrap)
{
  char name[MAX_NAME_LEN];
  char *p = name;
  int c = source_get();
  unsigned char sector = current_sector;
  Arglist * head = NULL;
  Arglist ** tail = &head;

  if (c == '+')
  {
    sector = 0;
    c = source_get();
  }
  while (c == ' ' || c == '\t')
    c = source_get();
  while (c != EOF) {
    switch (c) {
      case '\t':
      case ' ':  *p++ = ' ';
                 do
                   c = source_get();
                 while (c == ' ' || c == '\t');
                 break;
      default:
         if (c==nw_char)
           {
             ⟨ Look for end of scrap name and return 131 ⟩
             break;
           }
         if (!isgraph(c)) {
                  fprintf(stderr,
                          "%s: unexpected character in fragment name (%s, %d)\n",
                          command_name, source_name, source_line);
                  exit(-1);
                }
                *p++ = c;
                c = source_get();
                break;
    }
  }
  fprintf(stderr, "%s: unexpected end of file (%s, %d)\n",
          command_name, source_name, source_line);
  exit(-1);
  return NULL;  /* unreachable return to avoid warnings on some compilers */
}
◇
```

File defined by 14c, 118b, 119ab, 120b, 121a, 122, 124, 126, 129ac, 130, 133c, 134a.
Defines: `collect_scrap_name` 59, 78a, 101c, 118a, 133a.
Uses: `Arglist` 129b, `command_name` 17d, `current_sector` 27c, `exit` 11, `fprintf` 11, `isgraph` 11, `MAX_NAME_LEN` 12b, `nw_char` 17bc, `source_get` 87c, `source_line` 86b, `source_name` 86b, `stderr` 11.

⟨ *Look for end of scrap name and return* 131 ⟩ ≡

```
{
  Name * node;

  c = source_get();
  switch (c) {

    case '\'': {
          ⟨ Add plain string argument 132a ⟩
        }
        *p++ = ARG_CHR;
        c = source_get();
        break;
    case '1': case '2': case '3':
    case '4': case '5': case '6':
    case '7': case '8': case '9': {
          ⟨ Add a propagated argument 132b ⟩
        }
        *p++ = ARG_CHR;
        c = source_get();
        break;
    case '{': {
        ⟨ Add an inline scrap argument 132c ⟩
        }
        *p++ = ARG_CHR;
        c = source_get();
        break;
    case '<':
        ⟨ Add macro call argument 133a ⟩
        *p++ = ARG_CHR;
        c = source_get();
        break;
    case '(':
        scrap_name_has_parameters = 1;
        ⟨ Cleanup and install name 128b ⟩
        return buildArglist(node, head);
    case '>':
        scrap_name_has_parameters = 0;
        ⟨ Cleanup and install name 128b ⟩
        return buildArglist(node, head);

    default:
       if (c==nw_char)
         {
           *p++ = c;
              c = source_get();
              break;
         }
       fprintf(stderr,
                      "%s: unexpected %c%c in fragment invocation name (%s, %d)\n",
                      command_name, nw_char, c, source_name, source_line);
              exit(-1);
  }
}◇
```

Fragment referenced in 130.

Uses: ARG_CHR 127c, buildArglist 129c, command_name 17d, exit 11, fprintf 11, Name 117a, nw_char 17bc, source_get 87c, source_line 86b, source_name 86b, stderr 11.

A plain string argument has no name and a string as a value.

⟨ *Add plain string argument* 132a ⟩ ≡
```
char buff[MAX_NAME_LEN];
char * s = buff;
int c, c2;

while ((c = source_get()) != EOF) {
  if (c==nw_char) {
    c2 = source_get();
    if (c2==’\’’)
      break;
    *s++ = c2;
  }
  else
    *s++ = c;
}
*s = ’\000’;
```
⟨ *Add buff to current arg list* 133b ⟩◇

Fragment referenced in 131.
Uses: MAX_NAME_LEN 12b, nw_char 17bc, source_get 87c.

A parameter argument propagated into an interior fragment has no name and and a string of ARG_CHAR followed by a digit as value.

⟨ *Add a propagated argument* 132b ⟩ ≡
```
char buff[3];
buff[0] = ARG_CHR;
buff[1] = c;
buff[2] = ’\000’;
```
⟨ *Add buff to current arg list* 133b ⟩◇

Fragment referenced in 131.
Uses: ARG_CHR 127c.

⟨ *Add an inline scrap argument* 132c ⟩ ≡
```
int s = collect_scrap();
Scrap_Node * d = (Scrap_Node *)arena_getmem(sizeof(Scrap_Node));
d->scrap = s;
d->quoted = 0;
d->next = NULL;
*tail = buildArglist((Name *)1, (Arglist *)d);
tail = &(*tail)->next;◇
```

Fragment referenced in 131.
Uses: arena_getmem 152a, Arglist 129b, buildArglist 129c, collect_scrap 96b, Name 117a, Scrap_Node 116d.

⟨ *Type declarations* 132d ⟩ ≡
```
typedef struct embed {
   Scrap_Node * defs;
   Arglist * args;
} Embed_Node;
◇
```

Fragment defined by 12a, 83b, 116d, 117a, 127c, 129b, 132d, 144c, 150c.
Fragment referenced in 10.
Defines: Embed_Node 35, 104b, 113c.
Uses: Arglist 129b, Scrap_Node 116d.

⟨ *Add macro call argument* 133a ⟩ ≡

```
*tail = collect_scrap_name(current_scrap);
if (current_scrap >= 0)
  add_to_use((*tail)->name, current_scrap);
tail = &(*tail)->next;
◇
```

Fragment referenced in 131.
Uses: `add_to_use` 102a, `collect_scrap_name` 130.

⟨ *Add buff to current arg list* 133b ⟩ ≡

```
*tail = buildArglist(NULL, (Arglist *)save_string(buff));
tail = &(*tail)->next;
◇
```

Fragment referenced in 132ab.
Uses: `Arglist` 129b, `buildArglist` 129c, `save_string` 119a.

"names.c" 133c≡

```
static Scrap_Node *reverse(); /* a forward declaration */

void reverse_lists(names)
     Name *names;
{
  while (names) {
    reverse_lists(names->llink);
    names->defs = reverse(names->defs);
    names->uses = reverse(names->uses);
    names = names->rlink;
  }
}
◇
```

File defined by 14c, 118b, 119ab, 120b, 121a, 122, 124, 126, 129ac, 130, 133c, 134a.
Defines: `reverse_lists` 34d, 118a.
Uses: `Name` 117a, `reverse` 134a, `Scrap_Node` 116d.

Just for fun, here's a non-recursive version of the traditional list reversal code. Note that it reverses the list in place; that is, it does no new allocations.

"names.c" 134a≡

```
static Scrap_Node *reverse(a)
     Scrap_Node *a;
{
  if (a) {
    Scrap_Node *b = a->next;
    a->next = NULL;
    while (b) {
      Scrap_Node *c = b->next;
      b->next = a;
      a = b;
      b = c;
    }
  }
  return a;
}
◇
```

File defined by 14c, 118b, 119ab, 120b, 121a, 122, 124, 126, 129ac, 130, 133c, 134a.
Defines: `reverse` 133c.
Uses: `Scrap_Node` 116d.

3.4 Searching for Index Entries

Given the array of scraps and a set of index entries, we need to search all the scraps for occurrences of each entry. The obvious approach to this problem would be quite expensive for large documents; however, there is an interesting paper describing an efficient solution [**?**].

"scraps.c" 134b≡

```
typedef struct name_node {
  struct name_node *next;
  Name *name;
} Name_Node;
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `Name_Node` 134c, 140, 141, 142.
Uses: `Name` 117a.

"scraps.c" 134c≡

```
typedef struct goto_node {
  Name_Node *output;            /* list of words ending in this state */
  struct move_node *moves;      /* list of possible moves */
  struct goto_node *fail;       /* and where to go when no move fits */
  struct goto_node *next;       /* next goto node with same depth */
} Goto_Node;
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `Goto_Node` 135abc, 139b, 140, 141, 142.
Uses: `Name_Node` 134b.

"scraps.c" 135a≡

```
typedef struct move_node {
  struct move_node *next;
  Goto_Node *state;
  char c;
} Move_Node;
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `Move_Node` 135c, 140, 141.
Uses: `Goto_Node` 134c.

"scraps.c" 135b≡

```
static Goto_Node *root[128];
static int max_depth;
static Goto_Node **depths;
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `depths` 139b, 140, 141, `max_depth` 139b, 140, 141, `root` 139b, 140, 141, 142, 144b.
Uses: `Goto_Node` 134c.

"scraps.c" 135c≡

```
static Goto_Node *goto_lookup(c, g)
     char c;
     Goto_Node *g;
{
  Move_Node *m = g->moves;
  while (m && m->c != c)
    m = m->next;
  if (m)
    return m->state;
  else
    return NULL;
}
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `goto_lookup` 140, 141, 142.
Uses: `Goto_Node` 134c, `Move_Node` 135a.

3.4.1 Retrieving scrap uses

"scraps.c" 135d≡
```
typedef struct ArgMgr_s
{
   char * pv;
   char * bgn;
   Arglist * arg;
   struct ArgMgr_s * old;
} ArgMgr;
◇
```
File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `ArgMgr` 136abc, 138a.
Uses: `Arglist` 129b.

"scraps.c" 136a≡
```
typedef struct ArgManager_s
{
   Manager * m;
   ArgMgr * a;
} ArgManager;
◇
```
File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `ArgManager` 136bc, 138a, 142, 148a.
Uses: `ArgMgr` 135d, `Manager` 95c.

"scraps.c" 136b≡
```
static void
pushArglist(ArgManager * mgr, Arglist * a)
{
   ArgMgr * b = malloc(sizeof(ArgMgr));

   if (b == NULL)
   {
      fprintf(stderr, "Can't allocate space for an argument manager\n");
      exit(EXIT_FAILURE);
   }
   b->pv = b->bgn = NULL;
   b->arg = a;
   b->old = mgr->a;
   mgr->a = b;
}
◇
```
File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `pushArglist` 137b, 143b.
Uses: `Arglist` 129b, `ArgManager` 136a, `ArgMgr` 135d, `exit` 11, `fprintf` 11, `malloc` 11, `stderr` 11.

"scraps.c" 136c≡
```
static char argpop(ArgManager * mgr)
{
   while (mgr->a != NULL)
   {
      ArgMgr * a = mgr->a;

      ⟨ Perhaps —return— a character from the current arg 137a ⟩
      ⟨ Perhaps start a new arg 137b ⟩
      ⟨ Otherwise pop the current arg 137c ⟩
   }

   return (pop(mgr->m));
}
◇
```
File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: argpop 142, 143ab.
Uses: ArgManager 136a, ArgMgr 135d, pop 102c.

We separate individual arguments using spaces.

⟨ Perhaps —return— a character from the current arg 137a ⟩ ≡
```
if (a->pv != NULL)
{
   char c = *a->pv++;

   if (c != '\0')
      return c;
   a->pv = NULL;
   return ' ';
}
◇
```
Fragment referenced in 136c.

I'm pretty sure that only the first case is ever used. I don't think the others can occur in this context, which makes the whole —ArgManager— thing doubtful.

⟨ Perhaps start a new arg 137b ⟩ ≡
```
if (a->arg) {
   Arglist * b = a->arg;

   a->arg = b->next;
   if (b->name == NULL) {
      a->bgn = a->pv = (char *)b->args;
   } else if (b->name == (Name *)1) {
      a->bgn = a->pv = "{Embedded Scrap}";
   } else {
      pushArglist(mgr, b->args);
   }◇
```
Fragment referenced in 136c.
Uses: Arglist 129b, Name 117a, pushArglist 136b.

⟨ *Otherwise pop the current arg* 137c ⟩ ≡

```
} else {
   mgr->a = a->old;
   free(a);
}◇
```

Fragment referenced in 136c.

"scraps.c" 138a≡

```
static char
prev_char(ArgManager * mgr, int n)
{
   char c = '\0';
   ArgMgr * a = mgr->a;
   Manager * m = mgr->m;

   if (a != NULL) {
      ⟨ Get the nth previous character from an argument 138b ⟩
   } else {
      ⟨ Get the nth previous character from a scrap 138c ⟩
   }

   return c;
}
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `prev_char` 148a.
Uses: `ArgManager` 136a, `ArgMgr` 135d, `Manager` 95c.

⟨ *Get the nth previous character from an argument* 138b ⟩ ≡

```
if (a->pv && a->pv - n >= a->bgn)
   c = *a->pv;
else if (a->bgn) {
   int j = strlen(a->bgn) + 1;

   if (n >= j)
      c = a->bgn[j - n];
   else
      c = ' ';
}
◇
```

Fragment referenced in 138a.
Uses: `strlen` 11.

⟨ *Get the nth previous character from a scrap* 138c ⟩ ≡

```
int k = m->index - n - 2;

if (k >= 0)
   c = m->scrap->chars[k];
else if (m->prev)
   c = m->prev->chars[SLAB_SIZE - k];
◇
```

Fragment referenced in 138a.
Uses: `SLAB_SIZE` 92b.

3.4.2 Building the Automata

⟨ *Function prototypes* 139a ⟩ ≡

```
extern void search();
◇
```

Fragment defined by 25a, 39b, 52b, 55b, 69a, 82b, 86a, 93b, 102b, 114b, 118a, 128d, 139a, 148d, 151b.
Fragment referenced in 10.
Uses: search 139b.

"scraps.c" 139b≡

```
static void build_gotos();
static int reject_match();

void search()
{
  int i;
  for (i=0; i<128; i++)
    root[i] = NULL;
  max_depth = 10;
  depths = (Goto_Node **) arena_getmem(max_depth * sizeof(Goto_Node *));
  for (i=0; i<max_depth; i++)
    depths[i] = NULL;
  build_gotos(user_names);
  ⟨ Build failure functions 141 ⟩
  ⟨ Search scraps 142 ⟩
}
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: search 25b, 139a, 154.
Uses: arena_getmem 152a, build_gotos 139c, depths 135b, Goto_Node 134c, max_depth 135b, reject_match 148a, root 135b, user_names 117b.

"scraps.c" 139c≡

```
static void build_gotos(tree)
     Name *tree;
{
  while (tree) {
    ⟨ Extend goto graph with tree->spelling 140 ⟩
    build_gotos(tree->rlink);
    tree = tree->llink;
  }
}
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: build_gotos 139b.
Uses: Name 117a.

⟨ *Extend goto graph with* `tree->spelling` 140 ⟩ ≡

```
{
  int depth = 2;
  char *p = tree->spelling;
  char c = *p++;
  Goto_Node *q = root[c];
  Name_Node * last;
  if (!q) {
    q = (Goto_Node *) arena_getmem(sizeof(Goto_Node));
    root[c] = q;
    q->moves = NULL;
    q->fail = NULL;
    q->moves = NULL;
    q->output = NULL;
    q->next = depths[1];
    depths[1] = q;
  }
  while (c = *p++) {
    Goto_Node *new = goto_lookup(c, q);
    if (!new) {
      Move_Node *new_move = (Move_Node *) arena_getmem(sizeof(Move_Node));
      new = (Goto_Node *) arena_getmem(sizeof(Goto_Node));
      new->moves = NULL;
      new->fail = NULL;
      new->moves = NULL;
      new->output = NULL;
      new_move->state = new;
      new_move->c = c;
      new_move->next = q->moves;
      q->moves = new_move;
      if (depth == max_depth) {
        int i;
        Goto_Node **new_depths =
            (Goto_Node **) arena_getmem(2*depth*sizeof(Goto_Node *));
        max_depth = 2 * depth;
        for (i=0; i<depth; i++)
          new_depths[i] = depths[i];
        depths = new_depths;
        for (i=depth; i<max_depth; i++)
          depths[i] = NULL;
      }
      new->next = depths[depth];
      depths[depth] = new;
    }
    q = new;
    depth++;
  }
  last = q->output;
  q->output = (Name_Node *) arena_getmem(sizeof(Name_Node));
  q->output->next = last;
  q->output->name = tree;
}◇
```

Fragment referenced in 139c.

Uses: `arena_getmem` 152a, `depths` 135b, `goto_lookup` 135c, `Goto_Node` 134c, `max_depth` 135b, `Move_Node` 135a, `Name_Node` 134b, `root` 135b.

⟨ *Build failure functions* 141 ⟩ ≡

```
{
  int depth;
  for (depth=1; depth<max_depth; depth++) {
    Goto_Node *r = depths[depth];
    while (r) {
      Move_Node *m = r->moves;
      while (m) {
        char a = m->c;
        Goto_Node *s = m->state;
        Goto_Node *state = r->fail;
        while (state && !goto_lookup(a, state))
          state = state->fail;
        if (state)
          s->fail = goto_lookup(a, state);
        else
          s->fail = root[a];
        if (s->fail) {
          Name_Node *p = s->fail->output;
          while (p) {
            Name_Node *q = (Name_Node *) arena_getmem(sizeof(Name_Node));
            q->name = p->name;
            q->next = s->output;
            s->output = q;
            p = p->next;
          }
        }
        m = m->next;
      }
      r = r->next;
    }
  }
}◇
```

Fragment referenced in 139b.
Uses: `arena_getmem` 152a, `depths` 135b, `goto_lookup` 135c, `Goto_Node` 134c, `max_depth` 135b, `Move_Node` 135a, `Name_Node` 134b, `root` 135b.

3.4.3 Searching the Scraps

⟨ *Search scraps* 142 ⟩ ≡

```
{
  for (i=1; i<scraps; i++) {
    char c, last = '\0';
    Manager rd;
    ArgManager reader;
    Goto_Node *state = NULL;
    rd.prev = NULL;
    rd.scrap = scrap_array(i).slab;
    rd.index = 0;
    reader.m = &rd;
    reader.a = NULL;
    c = argpop(&reader);
    while (c) {
      while (state && !goto_lookup(c, state))
        state = state->fail;
      if (state)
        state = goto_lookup(c, state);
      else
        state = root[c];
      ⟨ Skip over at at 143a ⟩
      ⟨ Skip over a scrap use 143b ⟩
      ⟨ Skip over a block comment 34b ⟩
      last = c;
      c = argpop(&reader);
      if (state && state->output) {
        Name_Node *p = state->output;
        do {
          Name *name = p->name;
          if (!reject_match(name, c, &reader) &&
              scrap_array(i).sector == name->sector &&
              (!name->uses || name->uses->scrap != i)) {
            Scrap_Node *new_use =
                (Scrap_Node *) arena_getmem(sizeof(Scrap_Node));
            new_use->scrap = i;
            new_use->next = name->uses;
            name->uses = new_use;
            if (!scrap_is_in(name->defs, i))
              add_uses(&(scrap_array(i).uses), name);
          }
          p = p->next;
        } while (p);
      }
    }
  }
}◇
```

Fragment referenced in 139b.

Uses: `add_uses` 144b, `arena_getmem` 152a, `ArgManager` 136a, `argpop` 136c, `goto_lookup` 135c, `Goto_Node` 134c, `Manager` 95c, `Name` 117a, `Name_Node` 134b, `reject_match` 148a, `root` 135b, `scraps` 93a, `scrap_array` 93a, `scrap_is_in` 144a, `Scrap_Node` 116d.

⟨ *Skip over at at* 143a ⟩ ≡

```
if (last == nw_char && c == nw_char)
{
   last = '\0';
   c = argpop(&reader);
}
◇
```

Fragment referenced in 142.
Uses: `argpop` 136c, `nw_char` 17bc.

⟨ *Skip over a scrap use* 143b ⟩ ≡

```
if (last == nw_char && c == '<')
{
   char buf[MAX_NAME_LEN];
   char * p = buf;
   Arglist * args;

   c = argpop(&reader);
   while ((c = argpop(&reader)) != nw_char)
      *p++ = c;
   c = argpop(&reader);
   *p = '\0';
   if (sscanf(buf, "%p", &args) != 1) {
      fprintf(stderr,  "%s: found an internal problem (3)\n", command_name);
      exit(-1);
   }
   pushArglist(&reader, args);
}◇
```

Fragment referenced in 142.
Uses: `Arglist` 129b, `argpop` 136c, `command_name` 17d, `exit` 11, `fprintf` 11, `MAX_NAME_LEN` 12b, `nw_char` 17bc, `pushArglist` 136b, `stderr` 11.

⟨ *Forward declarations for scraps.c* 143c ⟩ ≡

```
static void add_uses();
static int scrap_is_in();
◇
```

Fragment defined by 106b, 113b, 125c, 143c.
Fragment referenced in 93a.
Uses: `add_uses` 144b, `scrap_is_in` 144a.

"scraps.c" 144a≡

```
static int scrap_is_in(Scrap_Node * list, int i)
{
  while (list != NULL) {
    if (list->scrap == i)
      return TRUE;
    list = list->next;
  }
  return FALSE;
}
```

◇

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `scrap_is_in` 142, 143c.
Uses: `FALSE` 12a, `Scrap_Node` 116d, `TRUE` 12a.

"scraps.c" 144b≡

```
static void add_uses(Uses * * root, Name *name)
{
   int cmp;
   Uses *p, **q = root;

   while ((p = *q, p != NULL)
          && (cmp = robs_strcmp(p->defn->spelling, name->spelling)) < 0)
      q = &(p->next);
   if (p == NULL || cmp > 0)
   {
      Uses *new = arena_getmem(sizeof(Uses));
      new->next = p;
      new->defn = name;
      *q = new;
   }
}
```

◇

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `add_uses` 101ab, 142, 143c.
Uses: `arena_getmem` 152a, `Name` 117a, `robs_strcmp` 121a, `root` 135b, `Uses` 144c.

⟨ *Type declarations* 144c ⟩ ≡

```
typedef struct uses {
  struct uses *next;
  Name *defn;
} Uses;
```

◇

Fragment defined by 12a, 83b, 116d, 117a, 127c, 129b, 132d, 144c, 150c.
Fragment referenced in 10.
Defines: `Uses` 41a, 144b, 145a, 146b.
Uses: `Name` 117a.

"scraps.c" 145a≡

```
void
format_uses_refs(FILE * tex_file, int scrap)
{
  Uses * p = scrap_array(scrap).uses;
  if (p != NULL)
    ⟨ Write uses references 145b ⟩
}
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: format_uses_refs 44ab.
Uses: FILE 11, scrap_array 93a, Uses 144c.

⟨ *Write uses references* 145b ⟩ ≡

```
{
  char join = ' ';
  fputs("\\item \\NWtxtIdentsUsed\\nobreak\\", tex_file);
  do {
    ⟨ Write one use reference 145c ⟩
    join = ',';
    p = p->next;
  }while (p != NULL);
  fputs(".", tex_file);
}◇
```

Fragment referenced in 145a.
Uses: fputs 11.

⟨ *Write one use reference* 145c ⟩ ≡

```
Name * name = p->defn;
Scrap_Node *defs = name->defs;
int first = TRUE, page = -1;
fprintf(tex_file,
        "%c \\verb%c%s%c\\nobreak\\ ",
        join, nw_char, name->spelling, nw_char);
if (defs)
{
  do {
    ⟨ Write one referenced scrap 146a ⟩
    first = FALSE;
    defs = defs->next;
  }while (defs!= NULL);
}
else
{
  fputs("\\NWnotglobal", tex_file);
}
◇
```

Fragment referenced in 145b.
Uses: FALSE 12a, first 151d, fprintf 11, fputs 11, Name 117a, nw_char 17bc, Scrap_Node 116d, TRUE 12a.

⟨ *Write one referenced scrap* 146a ⟩ ≡

```
fputs("\\NWlink{nuweb", tex_file);
write_scrap_ref(tex_file, defs->scrap, -1, &page);
fputs("}{", tex_file);
write_scrap_ref(tex_file, defs->scrap, first, &page);
fputs("}", tex_file);◇
```

Fragment referenced in 145c, 147a.
Uses: `first` 151d, `fputs` 11, `write_scrap_ref` 94a.

`"scraps.c"` 146b≡

```
void
format_defs_refs(FILE * tex_file, int scrap)
{
  Uses * p = scrap_array(scrap).defs;
  if (p != NULL)
    ⟨ Write defs references 146c ⟩
}
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `format_defs_refs` 44ab.
Uses: `FILE` 11, `scrap_array` 93a, `Uses` 144c.

⟨ *Write defs references* 146c ⟩ ≡

```
{
  char join = ' ';
  fputs("\\item \\NWtxtIdentsDefed\\nobreak\\", tex_file);
  do {
    ⟨ Write one def reference 147a ⟩
    join = ',';
    p = p->next;
  }while (p != NULL);
  fputs(".", tex_file);
}◇
```

Fragment referenced in 146b.
Uses: `fputs` 11.

⟨ *Write one def reference* 147a ⟩ ≡

```
Name * name = p->defn;
Scrap_Node *defs = name->uses;
int first = TRUE, page = -1;
fprintf(tex_file,
        "%c \\verb%c%s%c\\nobreak\\ ",
        join, nw_char, name->spelling, nw_char);
if (defs == NULL
    || (defs->scrap == scrap && defs->next == NULL)) {
  fputs("\\NWtxtIdentsNotUsed", tex_file);
}
else {
  do {
    if (defs->scrap != scrap) {
       ⟨ Write one referenced scrap 146a ⟩
       first = FALSE;
    }
    defs = defs->next;
  }while (defs!= NULL);
}
◇
```

Fragment referenced in 146c.
Uses: FALSE 12a, first 151d, fprintf 11, fputs 11, Name 117a, nw_char 17bc, Scrap_Node 116d, TRUE 12a.

Rejecting Matches

A problem with simple substring matching is that the string "he" would match longer strings like "she" and "her." Norman Ramsey suggested examining the characters occurring immediately before and after a match and rejecting the match if it appears to be part of a longer token. Of course, the concept of *token* is language-dependent, so we may be occasionally mistaken. For the present, we'll consider the mechanism an experiment.

"scraps.c" 147b≡

```
#define sym_char(c) (isalnum(c) || (c) == '_')

static int op_char(c)
     char c;
{
  switch (c) {
    case '!':           case '#': case '%': case '$': case '^':
    case '&': case '*': case '-': case '+': case '=': case '/':
    case '|': case '~': case '<': case '>':
      return TRUE;
    default:
      return c==nw_char ? TRUE : FALSE;
  }
}
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: op_char 148a, sym_char 148a.
Uses: FALSE 12a, nw_char 17bc, TRUE 12a.

"scraps.c" 148a≡

```
static int reject_match(name, post, reader)
     Name *name;
     char post;
     ArgManager *reader;
{
  int len = strlen(name->spelling);
  char first = name->spelling[0];
  char last = name->spelling[len - 1];
  char prev = prev_char(reader, len);
  if (sym_char(last) && sym_char(post)) return TRUE;
  if (sym_char(first) && sym_char(prev)) return TRUE;
  if (op_char(last) && op_char(post)) return TRUE;
  if (op_char(first) && op_char(prev)) return TRUE;
  return FALSE; /* Here is 148a-01 */
}
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `reject_match` 139b, 142.
Uses: `ArgManager` 136a, `FALSE` 12a, `first` 151d, `Name` 117a, `op_char` 147b, `prev_char` 138a, `strlen` 11, `sym_char` 147b, `TRUE` 12a.

3.5 Labels

Refer to 148b-01. And another one 148a-01.

⟨ *Get label from* 148b ⟩ ≡

```
char  label_name[MAX_NAME_LEN];
char * p = label_name;
while (c = @1, c != nw_char) /* Here is 148b-01 */
   *p++ = c;
*p = '\0';
c = @1;
◇
```

Fragment referenced in 57b, 99a, 110a.
Uses: `MAX_NAME_LEN` 12b, `nw_char` 17bc.

"scraps.c" 148c≡

```
void
write_label(char label_name[], FILE * file)
⟨ Search for label(⟨ Write the label to file 149a ⟩,⟨ Complain about missing label 149b ⟩) 150a ⟩
◇
```

File defined by 14b, 34e, 92bc, 93ac, 94ab, 95cd, 96ab, 102ac, 103ab, 104a, 105a, 106a, 115a, 134bc, 135abcd, 136abc, 138a, 139bc, 144ab, 145a, 146b, 147b, 148ac.
Defines: `write_label` 57b, 110a, 148d.
Uses: `FILE` 11.

⟨ *Function prototypes* 148d ⟩ ≡

```
void write_label(char label_name[], FILE * file);
◇
```

Fragment defined by 25a, 39b, 52b, 55b, 69a, 82b, 86a, 93b, 102b, 114b, 118a, 128d, 139a, 148d, 151b.
Fragment referenced in 10.
Uses: `FILE` 11, `write_label` 148c.

⟨ *Write the label to file* 149a ⟩ ≡

```
write_single_scrap_ref(file, lbl->scrap);
fprintf(file, "-%02d", lbl->seq);◇
```

Fragment referenced in 148c.
Uses: `fprintf` 11, `write_single_scrap_ref` 94b.

⟨ *Complain about missing label* 149b ⟩ ≡

```
fprintf(stderr, "Can't find label %s.\n", label_name);◇
```

Fragment referenced in 148c.
Uses: `fprintf` 11, `stderr` 11.

⟨ *Save label to label store* 149c ⟩ ≡

```
if (label_name[0])
⟨ Search for label(⟨ Complain about duplicate labels 149e ⟩,⟨ Create a new label entry 149d ⟩) 150a ⟩
else
{
    ⟨ Complain about empty label 149f ⟩
}◇
```

Fragment referenced in 99a.

⟨ *Create a new label entry* 149d ⟩ ≡

```
lbl = (label_node *)arena_getmem(sizeof(label_node) + (p - label_name));
lbl->left = lbl->right = NULL;
strcpy(lbl->name, label_name);
lbl->scrap = current_scrap;
lbl->seq = ++lblseq;
*plbl = lbl;◇
```

Fragment referenced in 149c.
Uses: `arena_getmem` 152a, `label_node` 150c.

⟨ *Complain about duplicate labels* 149e ⟩ ≡

```
fprintf(stderr, "Duplicate label %s.\n", label_name);◇
```

Fragment referenced in 149c.
Uses: `fprintf` 11, `stderr` 11.

⟨ *Complain about empty label* 149f ⟩ ≡

```
fprintf(stderr, "Empty label.\n");◇
```

Fragment referenced in 149c.
Uses: `fprintf` 11, `stderr` 11.

⟨ *Search for label(Found,Notfound)* 150a ⟩ ≡

```
{
   label_node * * plbl = &label_tab;
   for (;;)
   {
      label_node * lbl = *plbl;

      if (lbl)
      {
         int cmp = label_name[0] - lbl->name[0];

         if (cmp == 0)
            cmp = strcmp(label_name + 1, lbl->name + 1);
         if (cmp < 0)
            plbl = &lbl->left;
         else if (cmp > 0)
            plbl = &lbl->right;
         else
         {
            Found
            break;
         }
      }
      else
      {
          Notfound
          break;
      }
   }
}
```

◇

Fragment referenced in 148c, 149c.
Uses: `label_node` 150c, `label_tab` 150b.

"global.c" 150b≡

```
label_node * label_tab = NULL;
```

◇

File defined by 14e, 150b.
Defines: `label_tab` 150a, 151a.
Uses: `label_node` 150c.

⟨ *Type declarations* 150c ⟩ ≡

```
typedef struct l_node
{
   struct l_node * left, * right;
   int scrap, seq;
   char name[1];
} label_node;
```

◇

Fragment defined by 12a, 83b, 116d, 117a, 127c, 129b, 132d, 144c, 150c.
Fragment referenced in 10.
Defines: `label_node` 149d, 150ab, 151a.

⟨ *Global variable declarations* 151a ⟩ ≡

```
extern label_node * label_tab;
◇
```

Fragment defined by 16, 17bd, 27c, 32c, 48b, 86b, 95a, 117b, 151a.
Fragment referenced in 10.
Uses: `label_node` 150c, `label_tab` 150b.

3.6 Memory Management

I manage memory using a simple scheme inspired by Hanson's idea of *arenas* [**?**]. Basically, I allocate all the storage required when processing a source file (primarily for names and scraps) using calls to `arena_getmem(n)`, where `n` specifies the number of bytes to be allocated. When the storage is no longer required, the entire arena is freed with a single call to `arena_free()`. Both operations are quite fast.

⟨ *Function prototypes* 151b ⟩ ≡

```
extern void *arena_getmem();
extern void arena_free();
◇
```

Fragment defined by 25a, 39b, 52b, 55b, 69a, 82b, 86a, 93b, 102b, 114b, 118a, 128d, 139a, 148d, 151b.
Fragment referenced in 10.
Uses: `arena_free` 153, `arena_getmem` 152a.

`"arena.c"` 151c≡

```
typedef struct chunk {
  struct chunk *next;
  char *limit;
  char *avail;
} Chunk;
◇
```

File defined by 14d, 151cd, 152a, 153.
Defines: `Chunk` 151d, 152bc.

We define an empty chunk called `first`. The variable `arena` points at the current chunk of memory; it's initially pointed at `first`. As soon as some storage is required, a "real" chunk of memory will be allocated and attached to `first->next`; storage will be allocated from the new chunk (and later chunks if necessary).

`"arena.c"` 151d≡

```
static Chunk first = { NULL, NULL, NULL };
static Chunk *arena = &first;
◇
```

File defined by 14d, 151cd, 152a, 153.
Defines: `arena` 152abc, 153, `first` 94a, 145c, 146a, 147a, 148a, 153, 154.
Uses: `Chunk` 151c.

3.6.1 Allocating Memory

The routine `arena_getmem(n)` returns a pointer to (at least) `n` bytes of memory. Note that `n` is rounded up to ensure that returned pointers are always aligned. We align to the nearest 8-byte segment, since that'll satisfy the more common 2-byte and 4-byte alignment restrictions too.

"arena.c" 152a≡

```
void *arena_getmem(n)
     size_t n;
{
  char *q;
  char *p = arena->avail;
  n = (n + 7) & ~7;             /* ensuring alignment to 8 bytes */
  q = p + n;
  if (q <= arena->limit) {
    arena->avail = q;
    return p;
  }
  ⟨ Find a new chunk of memory 152b ⟩
}
◇
```

File defined by 14d, 151cd, 152a, 153.
Defines: arena_getmem 20b, 24a, 28b, 37, 93c, 95d, 96c, 101ab, 102a, 104bc, 119a, 123, 129c, 132c, 139b, 140, 141, 142, 144b, 149d, 151b.
Uses: arena 151d, size_t 11.

If the current chunk doesn't have adequate space (at least n bytes) we examine the rest of the list of chunks (starting at arena->next) looking for a chunk with adequate space. If n is very large, we may not find it right away or we may not find a suitable chunk at all.

⟨ Find a new chunk of memory 152b ⟩ ≡

```
{
  Chunk *ap = arena;
  Chunk *np = ap->next;
  while (np) {
    char *v = sizeof(Chunk) + (char *) np;
    if (v + n <= np->limit) {
      np->avail = v + n;
      arena = np;
      return v;
    }
    ap = np;
    np = ap->next;
  }
  ⟨ Allocate a new chunk of memory 152c ⟩
}◇
```

Fragment referenced in 152a.
Uses: arena 151d, Chunk 151c.

If there isn't a suitable chunk of memory on the free list, then we need to allocate a new one.

⟨ *Allocate a new chunk of memory* 152c ⟩ ≡

```
{
  size_t m = n + 10000;
  np = (Chunk *) malloc(m);
  np->limit = m + (char *) np;
  np->avail = n + sizeof(Chunk) + (char *) np;
  np->next = NULL;
  ap->next = np;
  arena = np;
  return sizeof(Chunk) + (char *) np;
}◇
```

Fragment referenced in 152b.
Uses: `arena` 151d, `Chunk` 151c, `malloc` 11, `size_t` 11.

3.6.2 Freeing Memory

To free all the memory in the arena, we need only point `arena` back to the first empty chunk.

"arena.c" 153≡

```
void arena_free()
{
  arena = &first;
}
◇
```

File defined by 14d, 151cd, 152a, 153.
Defines: `arena_free` 24b, 151b.
Uses: `arena` 151d, `first` 151d.

Chapter 4

Man page

Here is the UNIX man page for nuweb:

```
"nuweb.1" 154≡
      .TH NUWEB 1 "local 3/22/95"
      .SH NAME
      Nuweb, a literate programming tool
      .SH SYNOPSIS
      .B nuweb
      .br
      \fBnuweb\fP [options] [file] ...
      .SH DESCRIPTION
      .I Nuweb
      is a literate programming tool like Knuth's
      .I WEB,
      only simpler.
      A
      .I nuweb
      file contains program source code interleaved with documentation.
      When
      .I nuweb
      is given a
      .I nuweb
      file, it writes the program file(s),
      and also
      produces,
      .I LaTeX
      source for typeset documentation.
      .SH COMMAND LINE OPTIONS
      .br
      \fB-t\fP Suppresses generation of the {\tt .tex} file.
      .br
      \fB-o\fP Suppresses generation of the output files.
      .br
      \fB-d\fP List dangling identifier references in indexes.
      .br
      \fB-c\fP Forces output files to overwrite old files of the same
        name without comparing for equality first.
      .br
      \fB-v\fP The verbose flag. Forces output of progress reports.
      .br
      \fB-n\fP Forces sequential numbering of scraps (instead of page
        numbers).
```

```
.br
\fB-s\fP Doesn't print list of scraps making up file at end of
  each scrap.
\fB-p path\fP Prepend path to the filenames for all the output files.
\fB-V string\fP Provide the string for replacement of the @v
operation. This is intended as a means for including version
information in generated output.
\fB-x\fP Include cross-references in comments in output files.
\fB-h options\fP Turn on hyperlinks using the hyperref package of
LaTeX and provide the options to the package.
\fB-r\fP Turn on hyperlinks using the hyperref package of
LaTeX, with the package options being in the text.
\fB-I path\fP Provide a directory to search for included files. This
may appear several times.

.SH FORMAT OF NUWEB FILES
A
.I nuweb
file contains mostly ordinary
.I LaTeX.
The file is read and copied to output (.tex file) unless a
.I nuweb
command is encountered. All
.I nuweb
commands start with an ``at-sign'' (@).
Files and fragments are defined with the following commands:
.PP
@o \fIfile-name flags  scrap\fP  where scrap is smaller than one page.
.br
@O \fIfile-name flags  scrap\fP  where scrap is bigger than one page.
.br
@d \fIfragment-name scrap\fP. Where scrap is smallar than one page.
.br
@D \fIfragment-name scrap\fP. Where scrap is bigger than one page.
.PP
@q \fIfragment-name scrap\fP. Where scrap is smallar than one page.
The scrap is not expanded in the output, allowing you to construct
output files which can, perhaps after further processing, be input to
.I nuweb.
.br
@Q \fIfragment-name scrap\fP. Where scrap is bigger than one page.
Likewise.
.PP
Scraps have specific begin and end
markers;
which begin and end marker you use determines how the scrap will be
typeset in the .tex file:
.br
\fB@{\fP...\fB@}\fP for verbatim "terminal style" formatting or,
with the -l flag, LaTeX listing package.
.br
\fB@[\fP...\fB@]\fP for LaTeX paragraph mode formatting, and
.br
\fB@(\fP...\fB@)\fP for LaTeX math mode formmating.
.br
Any amount of whitespace
(including carriage returns) may appear between a name and the
begining of a scrap.
```

```
.PP
Several code/file scraps may have the same name;
.I nuweb
concatenates their definitions to produce a single scrap.
Code scrap definitions are like macro definitions;
.I nuweb
extracts a program by expanding one scrap.
The definition of that scrap contains references to other scraps, which are
themselves expanded, and so on.
\fINuweb\fP's output is readable; it preserves the indentation of expanded
scraps with respect to the scraps in which they appear.
.PP
.SH PER FILE OPTIONS
When defining an output file, the programmer has the option of using flags
to control the output.
.PP
\fB-d\fR option,
.I Nuweb
will emit line number indications at scrap boundaries.
.br
\fB-i\fR option,
.I Nuweb
supresses the indentation of fragments (useful for \fBFortran\fR).
.br
\fB-t\fP option makes \fInuweb\fP
copy tabs untouched from input to output.
.br
\fB-c\fIx\fP Include comments in the output file.
\fIx\fP may be \fBc\fP for C-style comments, \fB+\fP for C++ and
\fBp\fP for perl and similar.
.PP
.SH MINOR COMMANDS
.br
@@    Causes a single ``at-sign'' to be copied into the output.
.br
@\_    Causes the text between it and the next {\tt @\_} to be made bold
       (for keywords, etc.) in the formatted document
.br
@%     Comments out a line so that it doesn't appear in the output.
.br
@i     \fBfilename\fR causes the file named to be included.
.br
@f     Creates an index of output files.
.br
@m     Creates an index of fragments.
.br
@u     Creates an index of user-specified identifiers.
.PP
To mark an identifier for inclusion in the index, it must be mentioned
at the end of the scrap it was defined in. The line starts
with @| and ends with the \fBend of scrap\fP mark \fB@}\fP.
.PP
.SH ERROR MESSAGES
.PP
.SH BUGS
.PP
.SH AUTHOR
Preston Briggs.
```

```
Internet address \fBpreston@cs.rice.edu\fP.
.SH MAINTAINER
Simon Wright
Internet address \fBsimon@pushface.org\fP
.br
Keith Harwood
Internet address \fBKeith.Harwood@vitalmis.com\fP
◇
```

Uses: FILE 11, first 151d, scraps 93a, search 139b.

Chapter 5

Indices

Three sets of indices can be created automatically: an index of file names, an index of fragment names, and an index of user-specified identifiers. An index entry includes the name of the entry, where it was defined, and where it was referenced.

5.1 Files

5.2 Fragments

⟨ Begin the cross-reference environment 48d ⟩ Referenced in 44ab.
⟨ Begin the scrap environment 46a ⟩ Referenced in 44ab.
⟨ Bold Keyword 58a ⟩ Referenced in 56a.
⟨ Build a new arglist 104c ⟩ Referenced in 104a.
⟨ Build failure functions 141 ⟩ Referenced in 139b.
⟨ Build fragment definition 28a ⟩ Referenced in 26.
⟨ Build output file definition 27e ⟩ Referenced in 26.
⟨ Build `source_name` and `tex_name` 23 ⟩ Referenced in 22.
⟨ Check at-sequence for end-of-scrap 56a ⟩ Referenced in 54.
⟨ Check for ambiguous prefix 120a ⟩ Referenced in 119b.
⟨ Check for end of scrap name 105b ⟩ Referenced in 105a.
⟨ Check for macro invocation in scrap 109c ⟩ Referenced in 107.
⟨ Check for macro parameters 37 ⟩ Referenced in 105b.
⟨ Check for terminating at-sequence and return name 127a ⟩ Referenced in 126.
⟨ Check HTML at-sequence for end-of-scrap 77 ⟩ Referenced in 76.
⟨ Cleanup and install name 128b ⟩ Referenced in 127a, 128c, 131.
⟨ Close the current sector 27b ⟩ Referenced in 26, 42a.
⟨ Collect a block comment 29b ⟩ Referenced in 26.
⟨ Collect include-file name 91a ⟩ Referenced in 90.
⟨ Collect user-specified index entries 100 ⟩ Referenced in 98.
⟨ Comment this macro use 113a ⟩ Referenced in 111, 112ac.
⟨ Compare the temp file and the old file 85a ⟩ Referenced in 84b.
⟨ Complain about duplicate labels 149e ⟩ Referenced in 149c.
⟨ Complain about empty label 149f ⟩ Referenced in 149c.
⟨ Complain about missing label 149b ⟩ Referenced in 148c.
⟨ Copy block comment from scrap 33b ⟩ Referenced in 109c.
⟨ Copy file name into file 110c ⟩ Referenced in 109c.
⟨ Copy fragment title into file 112c ⟩ Referenced in 109c.
⟨ Copy label from scrap into file 110a ⟩ Referenced in 109c.
⟨ Copy label from source into 57b ⟩ Referenced in 42a, 56a.
⟨ Copy macro into `file` 111 ⟩ Referenced in 109c.
⟨ Copy version info into file 57a ⟩ Referenced in 56a, 109c.
⟨ Copy version info into tex file 43 ⟩ Referenced in 42a.
⟨ Copy `defs->scrap` to `file` 107 ⟩ Referenced in 106a.
⟨ Copy `source_file` into `html_file` 70b ⟩ Referenced in 70a.
⟨ Copy `source_file` into `tex_file` 41c ⟩ Referenced in 40b.
⟨ Create a new label entry 149d ⟩ Referenced in 149c.
⟨ Create new name entry 123 ⟩ Referenced in 119b, 122.
⟨ Create new scrap, managed by `writer` 96c ⟩ Referenced in 96b.
⟨ End block 46d ⟩ Referenced in 34c, 46a, 48a.
⟨ Enter the next argument 127b ⟩ Referenced in 127a.
⟨ Expand tab into spaces 55c ⟩ Referenced in 54, 76, 109b.
⟨ Extend goto graph with `tree->spelling` 140 ⟩ Referenced in 139c.
⟨ Fill in the middle of HTML scrap environment 73c ⟩ Referenced in 72ac.
⟨ Fill in the middle of the scrap environment 47 ⟩ Referenced in 44ab.
⟨ Find a free temporary file 84a ⟩ Referenced in 83c.
⟨ Find a new chunk of memory 152b ⟩ Referenced in 152a.
⟨ Finish HTML scrap environment 73d ⟩ Referenced in 72ac.
⟨ Finish the cross-reference environment 49a ⟩ Referenced in 44ab.
⟨ Finish the scrap environment 48a ⟩ Referenced in 44ab.
⟨ For all remaining scraps 116b ⟩ Referenced in 115b.
⟨ Format a file index entry 62c ⟩ Referenced in 62b.
⟨ Format a user HTML index entry 82a ⟩ Referenced in 81b.
⟨ Format a user index entry 68 ⟩ Referenced in 67c.
⟨ Format an HTML index entry 80a ⟩ Referenced in 79c.
⟨ Format an index entry 65b ⟩ Referenced in 64b.
⟨ Format HTML macro name 78a ⟩ Referenced in 77.
⟨ Format HTML macro parameters 39a ⟩ Referenced in 78a.

⟨ Format macro name 59 ⟩ Referenced in 56a.
⟨ Format macro parameters 38 ⟩ Referenced in 59, 61.
⟨ Forward declarations for scraps.c 106b, 113b, 125c, 143c ⟩ Referenced in 93a.
⟨ Function prototypes 25a, 39b, 52b, 55b, 69a, 82b, 86a, 93b, 102b, 114b, 118a, 128d, 139a, 148d, 151b ⟩ Referenced in 10.
⟨ Get comment delimiters 125b ⟩ Referenced in 125a.
⟨ Get label from 148b ⟩ Referenced in 57b, 99a, 110a.
⟨ Get label while collecting scrap 99a ⟩ Referenced in 98.
⟨ Get the nth previous character from a scrap 138c ⟩ Referenced in 138a.
⟨ Get the nth previous character from an argument 138b ⟩ Referenced in 138a.
⟨ Global variable declarations 16, 17bd, 27c, 32c, 48b, 86b, 95a, 117b, 151a ⟩ Referenced in 10.
⟨ Global variable definitions 17ac, 18a, 27d, 86c, 95b, 117c ⟩ Referenced in 14e.
⟨ Handle an “at” character 89 ⟩ Referenced in 87c.
⟨ Handle at-sign during scrap accumulation 98 ⟩ Referenced in 97.
⟨ Handle macro invocation in scrap 101c ⟩ Referenced in 98.
⟨ Handle macro parameter substitution 35 ⟩ Referenced in 109c.
⟨ Handle optional per-file flags 125a ⟩ Referenced in 124.
⟨ Handle tab characters on output 109b ⟩ Referenced in 107.
⟨ Handle the file name in `argv[arg]` 22 ⟩ Referenced in 21c.
⟨ Handle `EOF` 91b ⟩ Referenced in 87c.
⟨ If we break the line at this word 34a ⟩ Referenced in 33b.
⟨ Include a fragment use in comment 114a ⟩ Referenced in 113b.
⟨ Include an embedded scrap in comment 113c ⟩ Referenced in 113b.
⟨ Include block comment in a scrap 33a ⟩ Referenced in 98.
⟨ Include files 11 ⟩ Referenced in 10.
⟨ Indent suppressed 109a ⟩ Referenced in 108c.
⟨ Insert appropriate indentation 108c ⟩ Referenced in 107, 108b.
⟨ Insert debugging information if required 108b ⟩ Referenced in 107, 109c.
⟨ Interpret at-sequence 42a ⟩ Referenced in 41c.
⟨ Interpret command-line arguments 18bc ⟩ Referenced in 15a.
⟨ Interpret HTML at-sequence 71 ⟩ Referenced in 70b.
⟨ Interpret the argument string `s` 19 ⟩ Referenced in 18c.
⟨ Italic ”*whatever*” 58b ⟩ Referenced in 56a.
⟨ Limits 12b ⟩ Referenced in 10.
⟨ Look for end of scrap name and return 131 ⟩ Referenced in 130.
⟨ Make this argument 128a ⟩ Referenced in 127b.
⟨ Move a word to the file 33c ⟩ Referenced in 33b.
⟨ Move the temporary file to the target, if required 84b ⟩ Referenced in 83c.
⟨ Open an include file 90 ⟩ Referenced in 89.
⟨ Operating System Dependencies 15b ⟩ Referenced in 10, 14e.
⟨ Otherwise pop the current arg 137c ⟩ Referenced in 136c.
⟨ Perhaps add an include path 20b ⟩ Referenced in 18c.
⟨ Perhaps add letters to the page numbers 116c ⟩ Referenced in 115b.
⟨ Perhaps comment this macro 112a ⟩ Referenced in 35, 111.
⟨ Perhaps get the hyperref options 21a ⟩ Referenced in 18c.
⟨ Perhaps get the prepend path 20a ⟩ Referenced in 18c.
⟨ Perhaps get the version info string 20c ⟩ Referenced in 18c.
⟨ Perhaps put a delayed indent 112b ⟩ Referenced in 33b, 112a.
⟨ Perhaps —return— a character from the current arg 137a ⟩ Referenced in 136c.
⟨ Perhaps skip white-space 32a ⟩ Referenced in 31b.
⟨ Perhaps start a new arg 137b ⟩ Referenced in 136c.
⟨ Process a file 24b ⟩ Referenced in 22.
⟨ Process the remaining arguments (file names) 21c ⟩ Referenced in 15a.
⟨ Put out the indent 108d ⟩ Referenced in 35, 108c.
⟨ Rename the temporary file to the target 85b ⟩ Referenced in 84b, 85a.
⟨ Reverse cross-reference lists 34d ⟩ Referenced in 25b.
⟨ Rob’s ordering 64c ⟩ Referenced in 64b.
⟨ Save label to label store 149c ⟩ Referenced in 99a.
⟨ Save macro name 101d ⟩ Referenced in 101c.

⟨ Save macro parameters 36 ⟩ Referenced in 101c.
⟨ Scan at-sequence 26 ⟩ Referenced in 25c.
⟨ Scan the source file, looking for at-sequences 25c ⟩ Referenced in 25b.
⟨ Search for label(*Found*,*Notfound*) 150a ⟩ Referenced in 148c, 149c.
⟨ Search scraps 142 ⟩ Referenced in 139b.
⟨ Set locale information 21b ⟩ Referenced in 15a.
⟨ Set up name, args and next 104b ⟩ Referenced in 104a.
⟨ Show presence of a block comment 32d ⟩ Referenced in 56a.
⟨ Skip commented-out code 57e ⟩ Referenced in 56a, 77, 98.
⟨ Skip invisibles on *p* 121b ⟩ Referenced in 121a.
⟨ Skip over a block comment 34b ⟩ Referenced in 142.
⟨ Skip over a scrap use 143b ⟩ Referenced in 142.
⟨ Skip over an in-text scrap 28c ⟩ Referenced in 26.
⟨ Skip over at at 143a ⟩ Referenced in 142.
⟨ Skip over at-sign or go to skipped 29a ⟩ Referenced in 28c.
⟨ Skip over index entries 57d ⟩ Referenced in 56a, 77.
⟨ Skip to the next nw-char 32b ⟩ Referenced in 29b.
⟨ Skip until scrap begins, then return name 128c ⟩ Referenced in 126.
⟨ Skip whitespace 30a ⟩ Referenced in 29b.
⟨ Sort *key* of size *n* for *ordering* 65a ⟩ Referenced in 64b.
⟨ Start block 46b ⟩ Referenced in 34c, 46a.
⟨ Step *i* to the next valid scrap 116a ⟩ Referenced in 115b.
⟨ Step to next sector 27a ⟩ Referenced in 26, 42a.
⟨ Switch block 46c ⟩ Referenced in 46a.
⟨ Type declarations 12a, 83b, 116d, 117a, 127c, 129b, 132d, 144c, 150c ⟩ Referenced in 10.
⟨ Warn (only once) about needing to rerun after Latex 94c ⟩ Referenced in 94a, 115a.
⟨ Write abbreviated definition list 60 ⟩ Referenced in 59, 61.
⟨ Write defining scrap numbers 66a ⟩ Referenced in 65b.
⟨ Write defs references 146c ⟩ Referenced in 146b.
⟨ Write file defs 49b ⟩ Referenced in 44a.
⟨ Write file's defining scrap numbers 63a ⟩ Referenced in 62c.
⟨ Write HTML abbreviated definition list 78b ⟩ Referenced in 78a.
⟨ Write HTML bold tag or end 79b ⟩ Referenced in 77.
⟨ Write HTML defining scrap numbers 80c ⟩ Referenced in 80a.
⟨ Write HTML file defs 74a ⟩ Referenced in 72a.
⟨ Write HTML file's defining scrap numbers 80b ⟩ Referenced in 80a.
⟨ Write HTML index of file names 78c ⟩ Referenced in 71.
⟨ Write HTML index of macro names 79a ⟩ Referenced in 71.
⟨ Write HTML index of user-specified names 81a ⟩ Referenced in 71.
⟨ Write HTML macro declaration 73a ⟩ Referenced in 72c.
⟨ Write HTML macro definition 72c ⟩ Referenced in 71.
⟨ Write HTML macro defs 74b ⟩ Referenced in 72c.
⟨ Write HTML macro refs 74c ⟩ Referenced in 72c.
⟨ Write HTML output file declaration 72b ⟩ Referenced in 72a.
⟨ Write HTML output file definition 72a ⟩ Referenced in 71.
⟨ Write HTML referencing scrap numbers 80d ⟩ Referenced in 80a.
⟨ Write in-text scrap 48c ⟩ Referenced in 42a.
⟨ Write index of file names 62a ⟩ Referenced in 42a.
⟨ Write index of macro names 63b ⟩ Referenced in 42a.
⟨ Write index of user-specified names 67b ⟩ Referenced in 42a.
⟨ Write LaTeX limbo definitions 41a ⟩ Referenced in 40b, 70a.
⟨ Write macro definition 44b ⟩ Referenced in 42a.
⟨ Write macro defs 49c ⟩ Referenced in 44b.
⟨ Write macro refs 50a ⟩ Referenced in 44b.
⟨ Write one def reference 147a ⟩ Referenced in 146c.
⟨ Write one referenced scrap 146a ⟩ Referenced in 145c, 147a.
⟨ Write one use reference 145c ⟩ Referenced in 145b.
⟨ Write out `files->spelling` 83c ⟩ Referenced in 83a.

⟨ Write output file definition 44a ⟩ Referenced in 42a.
⟨ Write referencing scrap numbers 66b ⟩ Referenced in 65b.
⟨ Write the hyperlink usage macro 41b ⟩ Referenced in 41a.
⟨ Write the label to file 149a ⟩ Referenced in 148c.
⟨ Write the macro's name 45a ⟩ Referenced in 44b, 65b.
⟨ Write uses references 145b ⟩ Referenced in 145a.

5.3 Identifiers

Knuth prints his index of identifiers in a two-column format. I could force this automatically by emitting the `\twocolumn` command; but this has the side effect of forcing a new page. Therefore, it seems better to leave it this up to the user.