Nuweb Version 1.0b1
A Simple Literate Programming Tool

Preston Briggs[1]
preston@tera.com
HTML scrap generator by John D. Ramsdell
ramsdell@mitre.org
scrap formatting and continuing maintenance by Marc W. Mengel
mengel@fnal.gov

[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 [4]. 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 [5]) 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 [6, 7]. 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 [9]. Another system, FunnelWeb, is independent of any programming language and only mildly dependent on TEX [11]. 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 [8]. 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* [10]. 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.

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 *macros,* 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 macros are defined with the following commands:

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

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

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 macros may be similarly specified piecemeal.

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.

`@{`*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.

`@[`*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 macro.

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

`@<`*macro-name*`@( a1 @, a2 @) @>` Causes the macro *macro-name* to be expanded inline with the parameters *a1*, *a2*, etc. Up to 9 parameters may be given.

`@1, @2, ..., @9` In a macro causes the n'th macro parameter to be substituted into the scrap. If the parameter is not passed, a null string is substituted.

Note that macro names may be abbreviated, either during invocation or definition. For example, it would be very tedious to have to type, repeatedly, the macro 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 macro name uniquely, followed by three periods. An abbreviation may even occur before the full version; nuweb simply preserves the longest version of a macro name. Note also that blanks and tabs are insignificant within a macro 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 macro is expanded in a scrap, the body of the macro is indented to match the indentation of the macro 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 three "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 macros. That is, when a macro is expanded within a scrap, it will *not* be indented to match the indentation of the macro 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.

1.2.2 The Minor Commands

We have two 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.

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

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

`@f` Create an index of file names.

`@m` Create an index of macro name.

`@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 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.

1.4 Generating HTML

Nikos Drakos' LaTeX2HTML Version 0.5.3 [2] 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.

1.5 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 macros that haven't been defined, but don't halt. This seems most convenient for development, but may change in the future.
- File names and index entries should not contain any `@` signs.
- Macro 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, macros are displayed in a preformatted element (PRE), so macros 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.
- 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.

1.6 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, 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, macro 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" 1 ≡
    ⟨ Include files 2 ⟩
    ⟨ Type declarations 3, ... ⟩
    ⟨ Global variable declarations 16, ... ⟩
    ⟨ Function prototypes 29, ... ⟩
    ◇
```

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

⟨ Include files 2 ⟩ ≡

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

Macro referenced in 1.

I also 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` be default. The following code seems to work on all the local systems.

⟨ Type declarations 3 ⟩ ≡
```
#ifndef FALSE
#define FALSE 0
#endif
#ifndef TRUE
#define TRUE 1
#endif
◇
```
Macro defined by 3, 161, 162.
Macro referenced in 1.

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" 4 ≡
```
#include "global.h"
◇
```
File defined by 4, 14.

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

"pass1.c" 5 ≡
```
#include "global.h"
◇
```
File defined by 5, 30.

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" 6 ≡
```
#include "global.h"
static int scraps = 1;
◇
```
File defined by 6, 44, 45, 57, 58, 69, 75.

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" 7 ≡
```
#include "global.h"
static int scraps = 1;
◇
```
File defined by 7, 78, 79, 92, 93, 94, 95, 102, 108.

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

"output.c" 8 ≡
```
#include "global.h"
◇
```
File defined by 8, 111.

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" 9 ≡
     #include "global.h"
     ◇
```

File defined by 9, 118, 119, 120, 121, 126.

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

```
"scraps.c" 10 ≡
     #include "global.h"
     ◇
```

File defined by 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

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

```
"names.c" 11 ≡
     #include "global.h"
     ◇
```

File defined by 11, 166, 167, 168, 170, 171, 172, 174, 176, 180, 182, 183.

Memory allocation and deallocation is handled by routines in `arena.c` (see Section 3.5).

```
"arena.c" 12 ≡
     #include "global.h"
     ◇
```

File defined by 12, 198, 199, 200, 203.

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

```
"global.c" 13 ≡
     #include "global.h"
     ⟨ Global variable definitions 17, ... ⟩
     ◇
```

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" 14 ≡

     ⟨ Operating System Dependencies 15 ⟩
     int main(argc, argv)
          int argc;
          char **argv;
     {
       int arg = 1;
       ⟨ Avoid rename() problems 112 ⟩
       ⟨ Interpret command-line arguments 22, ... ⟩
       ⟨ Process the remaining arguments (file names) 25 ⟩
       exit(0);
     }
     ◇
```

File defined by 4, 14.

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 15 ⟩ ≡

```
#if defined(VMS)
#define PATH_SEP(c) (c==']'||c==':')
#elif defined(MSDOS)
#define PATH_SEP(c) (c=='\\')
#else
#define PATH_SEP(c) (c=='/')
#endif
#include <stdlib.h>
◇
```

Macro referenced in 14.

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.

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 flags */
◇
```

Macro defined by 16, 18, 20, 116, 135, 163.
Macro referenced in 1.

The flags are all initialized for correct default behavior.

⟨ Global variable definitions 17 ⟩ ≡

```
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;
◇
```

Macro defined by 17, 19, 21, 117, 136, 164.
Macro referenced in 13.

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

⟨ Global variable declarations 18 ⟩ ≡
```
extern int nw_char;
```
◇
Macro defined by 16, 18, 20, 116, 135, 163.
Macro referenced in 1.

⟨ Global variable definitions 19 ⟩ ≡
```
int nw_char='@';
```
◇
Macro defined by 17, 19, 21, 117, 136, 164.
Macro referenced in 13.

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

⟨ Global variable declarations 20 ⟩ ≡
```
extern char *command_name;
```
◇
Macro defined by 16, 18, 20, 116, 135, 163.
Macro referenced in 1.

⟨ Global variable definitions 21 ⟩ ≡
```
char *command_name = NULL;
```
◇
Macro defined by 17, 19, 21, 117, 136, 164.
Macro referenced in 13.

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

⟨ Interpret command-line arguments 22 ⟩ ≡
```
command_name = argv[0];
```
◇
Macro defined by 22, 23.
Macro referenced in 14.

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

⟨ Interpret command-line arguments 23 ⟩ ≡
```
while (arg < argc) {
  char *s = argv[arg];
  if (*s++ == '-') {
    ⟨ Interpret the argument string s 24 ⟩
    arg++;
  }
  else break;
}
```
◇
Macro defined by 22, 23.
Macro referenced in 14.

Several flags can be stacked behind a single minus sign; therefore, we've got to loop through the string, handling them all.

⟨ Interpret the argument string `s` 24 ⟩ ≡

```
{
  char c = *s++;
  while (c) {
    switch (c) {
      case 'c': compare_flag = FALSE;
                break;
      case 'd': dangling_flag = TRUE;
                break;
      case 'n': number_flag = TRUE;
                break;
      case 'o': output_flag = FALSE;
                break;
      case 's': scrap_flag = FALSE;
                break;
      case 't': tex_flag = FALSE;
                break;
      case 'v': verbose_flag = TRUE;
                break;
      default:  fprintf(stderr, "%s: unexpected argument ignored.  ",
                        command_name);
                fprintf(stderr, "Usage is: %s [-cnotv] file...\n",
                        command_name);
                break;
    }
    c = *s++;
  }
}◇
```

Macro referenced in 23.

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) 25 ⟩ ≡

```
{
  if (arg >= argc) {
    fprintf(stderr, "%s: expected a file name.  ", command_name);
    fprintf(stderr, "Usage is: %s [-cnotv] file-name...\n", command_name);
    exit(-1);
  }
  do {
    ⟨ Handle the file name in argv[arg] 26 ⟩
    arg++;
  } while (arg < argc);
}◇
```

Macro referenced in 14.

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]` 26 ⟩ ≡

```
{
  char source_name[100];
  char tex_name[100];
  char aux_name[100];
  ⟨ Build source_name and tex_name 27 ⟩
  ⟨ Process a file 28 ⟩
}◇
```

Macro referenced in 25.

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` 27 ⟩ ≡

```
{
  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++;
  }
  *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.tex", trim);
    sprintf(aux_name, "%s.aux", trim);
    *dot = '.';
  }
  else {
    sprintf(tex_name, "%s.tex", trim);
    sprintf(aux_name, "%s.aux", trim);
    *q++ = '.';
    *q++ = 'w';
    *q = '\0';
  }
}◇
```

Macro referenced in 26.

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 3.5). The sequential numbering of scraps is forced when generating HTML.

⟨ Process a file 28 ⟩ ≡

```
{
  pass1(source_name);
  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);
      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();
}◇
```

Macro referenced in 26.

2.3 Pass One

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

⟨ Function prototypes 29 ⟩ ≡
```
extern void pass1();
```
◇
Macro defined by 29, 43, 59, 77, 110, 115, 130, 158, 165, 189, 197.
Macro referenced in 1.

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 macro-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" 30 ≡
```
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 31 ⟩
  if (tex_flag)
    search();
  ⟨ Reverse cross-reference lists 36 ⟩
}
```
◇
File defined by 5, 30.

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 31 ⟩ ≡
```
{
  int c = source_get();
  while (c != EOF) {
    if (c == nw_char)
      ⟨ Scan at-sequence 32 ⟩
    c = source_get();
  }
}
```
◇
Macro referenced in 30.

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 32 ⟩ ≡

```
{
  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 33 ⟩
              break;
    case 'D':
    case 'd': ⟨ Build macro definition 34 ⟩
              break;
    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 @ sequence ignored (%s, line %d)\n",
                      command_name, source_name, source_line);
              break;
  }
}◇
```

Macro referenced in 31.

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 macro names.

⟨ Build output file definition 33 ⟩ ≡

```
{
  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 35 ⟩
}◇
```

Macro referenced in 32.

⟨ Build macro definition 34 ⟩ ≡

```
{
  Name *name = collect_macro_name();
  int scrap = collect_scrap();
  ⟨ Add scrap to name's definition list 35 ⟩
}◇
```

Macro referenced in 32.

Since a file or macro 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 35 ⟩ ≡

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

Macro referenced in 33, 34.

2.3.2 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 36 ⟩ ≡

```
{
  reverse_lists(file_names);
  reverse_lists(macro_names);
  reverse_lists(user_names);
}◇
```

Macro referenced in 30.

2.3.3 Dealing with macro parameters

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

"scraps.c" 37 ≡

```
typedef int *Parameters;
◇
```

File defined by 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

When we are copying a scrap to the output, we can then pull the nth string from the `Parameters` list when we see an `@1` `@2`, etc.

⟨ Handle macro parameter substitution 38 ⟩ ≡

```
case '1': case '2': case '3':
case '4': case '5': case '6':
case '7': case '8': case '9':
          if ( parameters && parameters[c - '1'] ) {
            Scrap_Node param_defs;
            param_defs.scrap = parameters[c - '1'];
            param_defs.next = 0;
            write_scraps(file, &param_defs, global_indent + indent,
                      indent_chars, debug_flag, tab_flag, indent_flag, 0);
          } else {
            /* ZZZ need error message here */
          }
          break;
◇
```

Macro referenced in 156.

Now onto actually parsing macro parameters from a call. We start off checking for macro 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 39 ⟩ ≡

```
{
  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 current scrap to name's uses 147 ⟩
  } while( scrap_ended_with == ',' );
  do
    c = source_get();
  while( ' ' == c );
  if (c == nw_char) {
    c = source_get();
  }
  if (c != '>') {
    /* ZZZ print error */;
  }
}◇
```

Macro referenced in 145.

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 40 ⟩ ≡

```
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;
}
```

◇

Macro referenced in 150.

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

⟨ Format macro parameters 41 ⟩ ≡

```
char sep;

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

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

  source_last = '{';
  copy_scrap(file);

  ++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();
}
```

◇

Macro referenced in 65.

⟨ Format HTML macro parameters 42 ⟩ ≡

```
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);

  ++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);
```

◇

Macro referenced in 97.

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 43 ⟩ ≡
```
extern void write_tex();
```
◇
Macro defined by 29, 43, 59, 77, 110, 115, 130, 158, 165, 189, 197.
Macro referenced in 1.

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

"latex.c" 44 ≡
```
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_user_entry();
```
◇
File defined by 6, 44, 45, 57, 58, 69, 75.

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" 45 ≡
```
void write_tex(file_name, tex_name)
     char *file_name;
     char *tex_name;
{
  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 46 ⟩
    ⟨ Copy source_file into tex_file 47 ⟩
    fclose(tex_file);
  }
  else
    fprintf(stderr, "%s: can't open %s\n", command_name, tex_name);
}
```
◇
File defined by 6, 44, 45, 57, 58, 69, 75.

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 46 ⟩ ≡

```
fputs("\\newcommand{\\NWtarget}[2]{#2}\n", tex_file);
fputs("\\newcommand{\\NWlink}[2]{#2}\n", tex_file);
fputs("\\newcommand{\\NWtxtMacroDefBy}{Macro defined by}\n", tex_file);
fputs("\\newcommand{\\NWtxtMacroRefIn}{Macro referenced in}\n", tex_file);
fputs("\\newcommand{\\NWtxtMacroNoRef}{Macro 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{\\NWsep}{${\\diamond}$}\n", tex_file);
◇
```

Macro referenced in 45, 79.

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` 47 ⟩ ≡

```
{
  int c = source_get();
  while (c != EOF) {
    if (c == nw_char)
      {
      ⟨ Interpret at-sequence 48 ⟩
      }
    else {
      putc(c, tex_file);
      c = source_get();
    }
  }
}◇
```

Macro referenced in 45.

⟨ Interpret at-sequence 48 ⟩ ≡

```
{
  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 49 ⟩
              break;
    case 'D': big_definition = TRUE;
    case 'd': ⟨ Write macro definition 50 ⟩
              break;
    case 'f': ⟨ Write index of file names 67 ⟩
              break;
    case 'm': ⟨ Write index of macro names 68 ⟩
              break;
    case 'u': ⟨ Write index of user-specified names 74 ⟩
              break;
    default:
          if (c==nw_char)
            putc(c, tex_file);
          c = source_get();
              break;
  }
}◇
```

Macro referenced in 47.

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 macro definition should look like this (perhaps modulo the scrap references):

```
\begin{flushleft} \small
\begin{minipage}{\linewidth} \label{scrap37}
$\langle$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 Macro referenced in scrap 17b.
\end{list}
\end{minipage}\\[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).

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

⟨ Write output file definition 49 ⟩ ≡

```
{
  Name *name = collect_file_name();
  ⟨ Begin the scrap environment 51 ⟩
  fprintf(tex_file, "\\verb%c\"%s\"%c\\nobreak\\ {\\footnotesize ", nw_char, name->spelling, nw_char);
  fputs("\\NWtarget{nuweb", tex_file);
  write_single_scrap_ref(tex_file, scraps);
  fputs("}{", tex_file);
  write_single_scrap_ref(tex_file, scraps++);
  fputs("}", tex_file);
  fputs(" }$\\equiv$\n", tex_file);
  ⟨ Fill in the middle of the scrap environment 52 ⟩
  if ( scrap_flag ) {
    ⟨ Write file defs 54 ⟩
  }
  ⟨ Finish the scrap environment 53 ⟩
}◇
```

Macro referenced in 48.

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

⟨ Write macro definition 50 ⟩ ≡

```
{
  Name *name = collect_macro_name();
  ⟨ Begin the scrap environment 51 ⟩
  fprintf(tex_file, "$\\langle\\,$%s\\nobreak\\ {\\footnotesize ", name->spelling);
  fputs("\\NWtarget{nuweb", tex_file);
  write_single_scrap_ref(tex_file, scraps);
  fputs("}{", tex_file);
  write_single_scrap_ref(tex_file, scraps++);
  fputs("}", tex_file);
  fputs("}$\\,\\rangle\\equiv$\n", tex_file);
  ⟨ Fill in the middle of the scrap environment 52 ⟩
  ⟨ Write macro defs 55 ⟩
  ⟨ Write macro refs 56 ⟩
  ⟨ Finish the scrap environment 53 ⟩
}◇
```

Macro referenced in 48.

⟨ Begin the scrap environment 51 ⟩ ≡

```
{
  fputs("\\begin{flushleft} \\small", tex_file);
  if (!big_definition)
    fputs("\n\\begin{minipage}{\\linewidth}", tex_file);
  fprintf(tex_file, " \\label{scrap%d}\n", scraps);
}◇
```

Macro referenced in 49, 50.

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 52 ⟩ ≡

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

Macro referenced in 49, 50.

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 53 ⟩ ≡

```
{
  if (!big_definition)
    fputs("\\end{minipage}\\\\[4ex]\n", tex_file);
  fputs("\\end{flushleft}\n", tex_file);
  do
    c = source_get();
  while (isspace(c));
}◇
```

Macro referenced in 49, 50.

Formatting Cross References

⟨ Write file defs 54 ⟩ ≡

```
{
  if (name->defs->next) {
    fputs("\\vspace{-1ex}\n", tex_file);
    fputs("\\footnotesize\\addtolength{\\baselineskip}{-1ex}\n", tex_file);
    fputs("\\begin{list}{}{\\setlength{\\itemsep}{-\\parsep}", tex_file);
    fputs("\\setlength{\\itemindent}{-\\leftmargin}}\n", tex_file);
    fputs("\\item \\NWtxtFileDefBy\\ ", tex_file);
    print_scrap_numbers(tex_file, name->defs);
    fputs("\\end{list}\n", tex_file);
  }
  else
    fputs("\\vspace{-2ex}\n", tex_file);
}◇
```

Macro referenced in 49.

⟨ Write macro defs 55 ⟩ ≡

```
{
  fputs("\\vspace{-1ex}\n", tex_file);
  fputs("\\footnotesize\\addtolength{\\baselineskip}{-1ex}\n", tex_file);
  fputs("\\begin{list}{}{\\setlength{\\itemsep}{-\\parsep}", tex_file);
  fputs("\\setlength{\\itemindent}{-\\leftmargin}}\n", tex_file);
  if (name->defs->next) {
    fputs("\\item \\NWtxtMacroDefBy\\ ", tex_file);
    print_scrap_numbers(tex_file, name->defs);
  }
}◇
```

Macro referenced in 50.

⟨ Write macro refs 56 ⟩ ≡

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

Macro referenced in 50.

"latex.c" 57 ≡

```
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 6, 44, 45, 57, 58, 69, 75.

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.

"latex.c" 58 ≡

```
static char *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 */
  { "$", "$", "@", "", "" },
};
int scrap_type = 0;

void update_delimit_scrap()
{
  static int been_here_before = 0;


  if (!been_here_before) {
     int i,j;
     /* make sure strings are writable first */
     for(i = 0; i < 3; i++) {
        for(j = 0; j < 5; j++) {
           delimit_scrap[i][j] = strdup(delimit_scrap[i][j]);
        }
     }
  }
  /* {}-mode begin */
  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;
  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;
}

static void copy_scrap(file)
     FILE *file;
{
  int indent = 0;
  int c;
  if (source_last == '{') scrap_type = 0;
  if (source_last == '[') scrap_type = 1;
  if (source_last == '(') scrap_type = 2;
  c = source_get();
  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);
                 fputs(delimit_scrap[scrap_type][4], file);
                 fputs("\n", file);
                 fputs(delimit_scrap[scrap_type][3], file);
                 fputs(delimit_scrap[scrap_type][0], file);
                 indent = 0;
                 break;
      case '\t': ⟨ Expand tab into spaces 60 ⟩
                 break;
      default:
         if (c==nw_char)
           {
             ⟨ Check at-sequence for end-of-scrap 61 ⟩
             break;
           }
         putc(c, file);
         indent++;
```

⟨ Function prototypes 59 ⟩ ≡

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

Macro defined by 29, 43, 59, 77, 110, 115, 130, 158, 165, 189, 197.
Macro referenced in 1.

⟨ Expand tab into spaces 60 ⟩ ≡

```
{
  int delta = 3 - (indent % 3);
  indent += delta;
  while (delta > 0) {
    putc(' ', file);
    delta--;
  }
}◇
```

Macro referenced in 58, 95, 155.

⟨ Check at-sequence for end-of-scrap 61 ⟩ ≡

```
{
  c = source_get();
  switch (c) {
    case '|': ⟨ Skip over index entries 62 ⟩
    case ',':
    case ')':
    case ']':
    case '}': fputs(delimit_scrap[scrap_type][1], file);
              return;
    case '<': ⟨ Format macro name 65 ⟩
              break;
    case '%': ⟨ Skip commented-out code 63 ⟩
              break;
    case '_': ⟨ Bold Keyword 64 ⟩
              break;
    case '1': case '2': case '3':
    case '4': case '5': case '6':
    case '7': case '8': case '9':
              fputs(delimit_scrap[scrap_type][1], file);
              fputc(nw_char, file);
              fputc(c,   file);
              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;
  }
}◇
```

Macro referenced in 58.

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 62 ⟩ ≡

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

Macro referenced in 61, 96.

⟨ Skip commented-out code 63 ⟩ ≡

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

Macro referenced in 61, 96, 143.

This scrap helps deal with bold keywords:

⟨ Bold Keyword 64 ⟩ ≡

```
{
  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);
}◇
```

Macro referenced in 61.

⟨ Format macro name 65 ⟩ ≡

```
{
  Name *name = collect_scrap_name();
  fputs(delimit_scrap[scrap_type][1],file);
  fprintf(file, "\\hbox{$\\langle\\,$%s\\nobreak\\ ", name->spelling);
  if (scrap_name_has_parameters) {
    ⟨ Format macro parameters 41 ⟩
  }
  fprintf(file, "{\\footnotesize ");
  if (name->defs)
    ⟨ Write abbreviated definition list 66 ⟩
  else {
    putc('?', file);
    fprintf(stderr, "%s: never defined <%s>\n",
            command_name, name->spelling);
  }
  fputs("}$\\,\\rangle$}", file);
  fputs(delimit_scrap[scrap_type][0], file);
}◇
```

Macro referenced in 61.

⟨ Write abbreviated definition list 66 ⟩ ≡

```
{
  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);
}◇
```

Macro referenced in 65.

2.4.2 Generating the Indices

⟨ Write index of file names 67 ⟩ ≡

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

Macro referenced in 48.

⟨ Write index of macro names 68 ⟩ ≡

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

Macro referenced in 48.

"latex.c" 69 ≡

```
static void format_entry(name, tex_file, file_flag)
     Name *name;
     FILE *tex_file;
     int file_flag;
{
  while (name) {
    format_entry(name->llink, tex_file, file_flag);
    ⟨ Format an index entry 70 ⟩
    name = name->rlink;
  }
}
◇
```

File defined by 6, 44, 45, 57, 58, 69, 75.

⟨ Format an index entry 70 ⟩ ≡

```
{
  fputs("\\item ", tex_file);
  if (file_flag) {
    fprintf(tex_file, "\\verb%c\"%s\"%c ", nw_char, name->spelling, nw_char);
    ⟨ Write file's defining scrap numbers 71 ⟩
  }
  else {
    fprintf(tex_file, "$\\langle\\,$%s\\nobreak\\ {\\footnotesize ", name->spelling);
    ⟨ Write defining scrap numbers 72 ⟩
    fputs("}$\\,\\rangle$ ", tex_file);
    ⟨ Write referencing scrap numbers 73 ⟩
  }
  putc('\n', tex_file);
}◇
```

Macro referenced in 69.

⟨ Write file's defining scrap numbers 71 ⟩ ≡

```
{
  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);
}◇
```

Macro referenced in 70.

⟨ Write defining scrap numbers 72 ⟩ ≡

```
{
  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);
}◇
```

Macro referenced in 70.

⟨ Write referencing scrap numbers 73 ⟩ ≡

```
{
  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);
}◇
```

Macro referenced in 70.

⟨ Write index of user-specified names 74 ⟩ ≡

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

Macro referenced in 48.

"latex.c" 75 ≡

```
static void format_user_entry(name, tex_file)
     Name *name;
     FILE *tex_file;
{
  while (name) {
    format_user_entry(name->llink, tex_file);
    ⟨ Format a user index entry 76 ⟩
    name = name->rlink;
  }
}
◇
```

File defined by 6, 44, 45, 57, 58, 69, 75.

⟨ Format a user index entry 76 ⟩ ≡

```
{
  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 (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);
  }
}◇
```

Macro referenced in 75.

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 77 ⟩ ≡
```
extern void write_html();
```
◇

Macro defined by 29, 43, 59, 77, 110, 115, 130, 158, 165, 189, 197.
Macro referenced in 1.

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

"html.c" 78 ≡
```
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_user_entry();
```
◇

File defined by 7, 78, 79, 92, 93, 94, 95, 102, 108.

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.

"html.c" 79 ≡
```
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 46 ⟩
  if (html_file) {
    if (verbose_flag)
      fprintf(stderr, "writing %s\n", html_name);
    source_open(file_name);
    ⟨ Copy source_file into html_file 80 ⟩
    fclose(html_file);
  }
  else
    fprintf(stderr, "%s: can't open %s\n", command_name, html_name);
}
```
◇

File defined by 7, 78, 79, 92, 93, 94, 95, 102, 108.

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.

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

⟨ Copy `source_file` into `html_file` 80 ⟩ ≡

```
{
  int c = source_get();
  while (c != EOF) {
    if (c == nw_char)
      ⟨ Interpret HTML at-sequence 81 ⟩
    else {
      putc(c, html_file);
      c = source_get();
    }
  }
}◇
```

Macro referenced in 79.

⟨ Interpret HTML at-sequence 81 ⟩ ≡

```
{
  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 82 ⟩
              break;
    case 'D':
    case 'd': ⟨ Write HTML macro definition 84 ⟩
              break;
    case 'f': ⟨ Write HTML index of file names 99 ⟩
              break;
    case 'm': ⟨ Write HTML index of macro names 100 ⟩
              break;
    case 'u': ⟨ Write HTML index of user-specified names 107 ⟩
              break;
    default:
          if (c==nw_char)
            putc(c, html_file);
          c = source_get();
              break;
  }
}◇
```

Macro referenced in 80.

2.5.1 Formatting Definitions

We go through only a little amount of effort to format a definition. The HTML for the previous macro 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>
Macro referenced in scrap <a href="#nuweb67">67</a>.
<br>
```

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

⟨ Write HTML output file definition 82 ⟩ ≡

```
{
  Name *name = collect_file_name();
  ⟨ Begin HTML scrap environment 86 ⟩
  ⟨ Write HTML output file declaration 83 ⟩
  scraps++;
  ⟨ Fill in the middle of HTML scrap environment 87 ⟩
  ⟨ Write HTML file defs 89 ⟩
  ⟨ Finish HTML scrap environment 88 ⟩
}◇
```

Macro referenced in 81.

⟨ Write HTML output file declaration 83 ⟩ ≡

```
  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);
◇
```

Macro referenced in 82.

⟨ Write HTML macro definition 84 ⟩ ≡

```
{
  Name *name = collect_macro_name();
  ⟨ Begin HTML scrap environment 86 ⟩
  ⟨ Write HTML macro declaration 85 ⟩
  scraps++;
  ⟨ Fill in the middle of HTML scrap environment 87 ⟩
  ⟨ Write HTML macro defs 90 ⟩
  ⟨ Write HTML macro refs 91 ⟩
  ⟨ Finish HTML scrap environment 88 ⟩
}◇
```

Macro referenced in 81.

I don't format a macro 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 macro

names that are recognized in preformatted text elements (PRE).

Modification 2001–02–15.: I'm interpreting the macro 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 85 ⟩ ≡
```
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);
```
◇

Macro referenced in 84.

⟨ Begin HTML scrap environment 86 ⟩ ≡
```
{
  fputs("\\begin{rawhtml}\n", html_file);
  fputs("<pre>\n", html_file);
}
```
◇

Macro referenced in 82, 84.

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

⟨ Fill in the middle of HTML scrap environment 87 ⟩ ≡
```
{
  copy_scrap(html_file);
  fputs("&lt;&gt;</pre>\n", html_file);
}
```
◇

Macro referenced in 82, 84.

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

⟨ Finish HTML scrap environment 88 ⟩ ≡
```
{
  fputs("\\end{rawhtml}\n", html_file);
  c = source_get(); /* Get rid of current at command. */
}
```
◇

Macro referenced in 82, 84.

Formatting Cross References

⟨ Write HTML file defs 89 ⟩ ≡
```
{
  if (name->defs->next) {
    fputs("\\end{rawhtml}\\NWtxtFileDefBy\\begin{rawhtml} ", html_file);
    print_scrap_numbers(html_file, name->defs);
    fputs("<br>\n", html_file);
  }
}
```
◇

Macro referenced in 82.

⟨ Write HTML macro defs 90 ⟩ ≡

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

Macro referenced in 84.

⟨ Write HTML macro refs 91 ⟩ ≡

```
{
  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);
}◇
```

Macro referenced in 84.

"html.c" 92 ≡

```
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 7, 78, 79, 92, 93, 94, 95, 102, 108.

"html.c" 93 ≡

```
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 7, 78, 79, 92, 93, 94, 95, 102, 108.

"html.c" 94 ≡

```
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 7, 78, 79, 92, 93, 94, 95, 102, 108.

Formatting a Scrap

We must translate HTML special keywords into entities in scraps.

"html.c" 95 ≡

```
static void copy_scrap(file)
     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 60⟩
                 break;
      default:
         if (c==nw_char)
           {
             ⟨Check HTML at-sequence for end-of-scrap 96⟩
             break;
           }
         putc(c, file);
                 indent++;
                 break;
    }
    c = source_get();
  }
}
◇
```

File defined by 7, 78, 79, 92, 93, 94, 95, 102, 108.

⟨ Check HTML at-sequence for end-of-scrap 96 ⟩ ≡

```
{
  c = source_get();
  switch (c) {
    case '|': ⟨ Skip over index entries 62 ⟩
    case ',':
    case '}':
    case ']':
    case ')': return;
    case '_': ⟨ Write HTML bold tag or end 101 ⟩
              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 97 ⟩
              break;
    case '%': ⟨ Skip commented-out code 63 ⟩
              break;
    default:
         if (c==nw_char)
           {
             fputc(c, file);
             break;
           }
          /* ignore these since pass1 will have warned about them */
              break;
  }
}◇
```

Macro referenced in 95.

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 97 ⟩ ≡

```
{
  Name *name = collect_scrap_name();
  fputs("&lt;\\end{rawhtml}", file);
  fputs(name->spelling, file);
  if (scrap_name_has_parameters) {
    ⟨ Format HTML macro parameters 42 ⟩
  }
  fputs("\\begin{rawhtml} ", file);
  if (name->defs)
    ⟨ Write HTML abbreviated definition list 98 ⟩
  else {
    putc('?', file);
    fprintf(stderr, "%s: never defined <%s>\n",
            command_name, name->spelling);
  }
  fputs("&gt;", file);
}◇
```

Macro referenced in 96.

⟨ Write HTML abbreviated definition list 98 ⟩ ≡

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

Macro referenced in 97.

2.5.2 Generating the Indices

⟨ Write HTML index of file names 99 ⟩ ≡

```
{
  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();
}◇
```

Macro referenced in 81.

⟨ Write HTML index of macro names 100 ⟩ ≡

```
{
  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();
}◇
```

Macro referenced in 81.

⟨ Write HTML bold tag or end 101 ⟩ ≡

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

Macro referenced in 96.

"html.c" 102 ≡

```
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 103 ⟩
    name = name->rlink;
  }
}
◇
```

File defined by 7, 78, 79, 92, 93, 94, 95, 102, 108.

⟨ Format an HTML index entry 103 ⟩ ≡

```
{
  fputs("<dt> ", html_file);
  if (file_flag) {
    fprintf(html_file, "<code>\"%s\"</code>\n<dd> ", name->spelling);
    ⟨ Write HTML file's defining scrap numbers 104 ⟩
  }
  else {
    fputs("&lt;\\end{rawhtml}", html_file);
    fputs(name->spelling, html_file);
    fputs("\\begin{rawhtml} ", html_file);
    ⟨ Write HTML defining scrap numbers 105 ⟩
    fputs("&gt;\n<dd> ", html_file);
    ⟨ Write HTML referencing scrap numbers 106 ⟩
  }
  putc('\n', html_file);
}◇
```

Macro referenced in 102.

⟨ Write HTML file's defining scrap numbers 104 ⟩ ≡

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

Macro referenced in 103.

⟨ Write HTML defining scrap numbers 105 ⟩ ≡

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

Macro referenced in 103.

⟨ Write HTML referencing scrap numbers 106 ⟩ ≡

```
{
  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);
}◇
```

Macro referenced in 103.

⟨ Write HTML index of user-specified names 107 ⟩ ≡

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

Macro referenced in 81.

"html.c" 108 ≡

```
static void format_user_entry(name, html_file)
     Name *name;
     FILE *html_file;
{
  while (name) {
    format_user_entry(name->llink, html_file);
    ⟨ Format a user HTML index entry 109 ⟩
    name = name->rlink;
  }
}
◇
```

File defined by 7, 78, 79, 92, 93, 94, 95, 102, 108.

⟨ Format a user HTML index entry 109 ⟩ ≡

```
{
  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);
  }
}◇
```

Macro referenced in 108.

2.6 Writing the Output Files

⟨ Function prototypes 110 ⟩ ≡

```
extern void write_files();
◇
```

Macro defined by 29, 43, 59, 77, 110, 115, 130, 158, 165, 189, 197.
Macro referenced in 1.

"output.c" 111 ≡

```
void write_files(files)
     Name *files;
{
  while (files) {
    write_files(files->llink);
    ⟨ Write out files->spelling 113 ⟩
    files = files->rlink;
  }
}
◇
```

File defined by 8, 111.

We call `tempnam`, causing it to create a file name in the current directory. This could cause a problem for `rename` if the eventual output file will reside on a different file system.

To avoid this, we used to set the environment variable `TMPDIR` to `"."` at the beginning of the program, but since we got rid of tempnam(), we no longer bother.

⟨ Avoid rename() problems 112 ⟩ ≡

◇

Macro referenced in 14.

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.

Note: I've modified this on 2001-02-15 for compilation for Win32 with Borland C++ (assuming `MSDOS` is defined). The second argument to `tempname` cannot be null in that system.

⟨ Write out `files->spelling` 113 ⟩ ≡

```
{
  static char temp_name[] = "nw000000";
  static int temp_name_count = 0;
  char indent_chars[500];
  int temp_file_fd;
  FILE *temp_file;

  for( temp_name_count = 0; temp_name_count < 10000; temp_name_count++) {
    sprintf(temp_name,"nw%06d", 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);
  }
  if (verbose_flag)
    fprintf(stderr, "writing %s [%s]\n", files->spelling, temp_name);
  write_scraps(temp_file, files->defs, 0, indent_chars,
               files->debug_flag, files->tab_flag, files->indent_flag, 0);
  fclose(temp_file);
  if (compare_flag)
    ⟨ Compare the temp file and the old file 114 ⟩
  else {
    remove(files->spelling);
    rename(temp_name, files->spelling);
  }
}◇
```

Macro referenced in 111.

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

⟨ Compare the temp file and the old file 114 ⟩ ≡

```
{
  FILE *old_file = fopen(files->spelling, "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(files->spelling);
      rename(temp_name, files->spelling);
    }
  }
  else
    rename(temp_name, files->spelling);
}◇
```

Macro referenced in 113.

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 115 ⟩ ≡
```
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. */
```
◇

Macro defined by 29, 43, 59, 77, 110, 115, 130, 158, 165, 189, 197.
Macro referenced in 1.

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

⟨ Global variable declarations 116 ⟩ ≡
```
extern char *source_name;  /* name of the current file */
extern int source_line;    /* current line in the source file */
```
◇

Macro defined by 16, 18, 20, 116, 135, 163.
Macro referenced in 1.

⟨ Global variable definitions 117 ⟩ ≡
```
char *source_name = NULL;
int source_line = 0;
```
◇

Macro defined by 17, 19, 21, 117, 136, 164.
Macro referenced in 13.

3.1.2 Local Declarations

"input.c" 118 ≡
```
static FILE *source_file;  /* the current input file */
static int source_peek;
static int double_at;
static int include_depth;
```
◇

File defined by 9, 118, 119, 120, 121, 126.

"input.c" 119 ≡

```
static struct {
  FILE *file;
  char *name;
  int line;
} stack[10];
◇
```

File defined by 9, 118, 119, 120, 121, 126.

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" 120 ≡

```
int source_last;
int source_get()
{
  int c;
  source_last = c = source_peek;
  switch (c) {
    case EOF:  ⟨ Handle EOF 125 ⟩
               return c;
    case '\n': source_line++;
    default:
           if (c==nw_char)
             {
               ⟨ Handle an "at" character 122 ⟩
               return c;
             }
           source_peek = getc(source_file);
               return c;
  }
}
◇
```

File defined by 9, 118, 119, 120, 121, 126.

source_ungetc pushes a read character back to the source_file.

"input.c" 121 ≡

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

File defined by 9, 118, 119, 120, 121, 126.

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 122 ⟩ ≡

```
{
  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 123 ⟩
                break;
      case 'f': case 'm': case 'u':
      case 'd': case 'o': case 'D': case 'O':
      case '{': case '}': case '<': case '>': case '|':
      case '(': case ')': case '[': case ']':
      case '%': case '_':
      case ':': case ',':
      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 @ sequence %c[%d] (%s, line %d)\n",
                     command_name, c, c, source_name, source_line);
             exit(-1);
    }
}◇
```

Macro referenced in 120.

⟨ Open an include file 123 ⟩ ≡

```
{
  char name[100];
  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 124 ⟩
  stack[include_depth].name = source_name;
  stack[include_depth].file = source_file;
  stack[include_depth].line = source_line + 1;
  include_depth++;
  source_line = 1;
  source_name = save_string(name);
  source_file = fopen(source_name, "r");
  if (!source_file) {
    fprintf(stderr, "%s: can't open include file %s\n",
     command_name, source_name);
    exit(-1);
  }
  source_peek = getc(source_file);
  c = source_get();
}◇
```

Macro referenced in 122.

⟨ Collect include-file name 124 ⟩ ≡

```
{
    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);
    }
}◇
```

Macro referenced in 123.

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` 125 ⟩ ≡

```
{
  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();
  }
}◇
```

Macro referenced in 120.

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" 126 ≡

```
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 9, 118, 119, 120, 121, 126.

3.2 Scraps

"scraps.c" 127 ≡

```
#define SLAB_SIZE 500

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

File defined by 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

"scraps.c" 128 ≡

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

File defined by 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

"scraps.c" 129 ≡

```
static ScrapEntry *SCRAP[256];

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

static int scraps;
◇
```

File defined by 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

⟨ Function prototypes 130 ⟩ ≡

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

Macro defined by 29, 43, 59, 77, 110, 115, 130, 158, 165, 189, 197.
Macro referenced in 1.

"scraps.c" 131 ≡

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

File defined by 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

"scraps.c" 132 ≡

```
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 134 ⟩
  }
  if (first>=0)
  *page = scrap_array(num).page;
}
◇
```

File defined by 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

"scraps.c" 133 ≡

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

File defined by 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

⟨ Warn (only once) about needing to rerun after Latex 134 ⟩ ≡

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

Macro referenced in 132, 159.

⟨ Global variable declarations 135 ⟩ ≡

```
extern int already_warned;
◇
```

Macro defined by 16, 18, 20, 116, 135, 163.
Macro referenced in 1.

⟨ Global variable definitions 136 ⟩ ≡

```
int already_warned = 0;
```

◇

Macro defined by 17, 19, 21, 117, 136, 164.
Macro referenced in 13.

"scraps.c" 137 ≡

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

◇

File defined by 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

"scraps.c" 138 ≡

```
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 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

"scraps.c" 139 ≡

```
static void pushs(s, manager)
     char *s;
     Manager *manager;
{
  while (*s)
    push(*s++, manager);
}
```

◇

File defined by 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

"scraps.c" 140 ≡

```
int collect_scrap()
{
  int current_scrap;
  Manager writer;
  ⟨ Create new scrap, managed by writer 141 ⟩
  ⟨ Accumulate scrap and return scraps++ 142 ⟩
}
```

◇

File defined by 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

⟨ Create new scrap, managed by `writer` 141 ⟩ ≡

```
{
  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;
  writer.scrap = scrap;
  writer.index = 0;
  current_scrap = scraps++;
}◇
```

Macro referenced in 140.

⟨ Accumulate scrap and return `scraps++` 142 ⟩ ≡

```
{
  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 143 ⟩
                break;
          }
        push(c, &writer);
                c = source_get();
                break;
    }
  }
}◇
```

Macro referenced in 140.

⟨ Handle at-sign during scrap accumulation 143 ⟩ ≡

```
{
  c = source_get();
  switch (c) {
    case '|': ⟨ Collect user-specified index entries 144 ⟩
    case ',':
    case ')':
    case ']':
    case '}': push('\0', &writer);
              scrap_ended_with = c;
              return current_scrap;
    case '<': ⟨ Handle macro invocation in scrap 145 ⟩
              break;
    case '%': ⟨ Skip commented-out code 63 ⟩
              /* emit line break to the output file to keep #line in sync. */
              push('\n', &writer);
              c = source_get();
              break;
    case '1': case '2': case '3':
    case '4': case '5': case '6':
    case '7': case '8': case '9':
              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 in scrap (%s, %d)\n",
                      command_name, c, source_name, source_line);
              exit(-1);
  }
}◇
```

Macro referenced in 142.

⟨ Collect user-specified index entries 144 ⟩ ≡

```
{
  do {
    char new_name[100];
    char *p = new_name;
    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';
      name = name_add(&user_names, new_name);
      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;
      }
    }
  } while (c != nw_char);
  c = source_get();
  if (c != '}' && c != ']' && c != ')' ) {
    fprintf(stderr, "%s: unexpected @%c in index entry (%s, %d)\n",
            command_name, c, source_name, source_line);
    exit(-1);
  }
}◇
```

Macro referenced in 143.

⟨ Handle macro invocation in scrap 145 ⟩ ≡

```
{
  Name *name = collect_scrap_name();
  ⟨ Save macro name 146 ⟩
  ⟨ Add current scrap to name's uses 147 ⟩
  if (scrap_name_has_parameters) {
    ⟨ Save macro parameters 39 ⟩
  }
  push(nw_char, &writer);
  push('>', &writer);
  c = source_get();
}◇
```

Macro referenced in 143.

⟨ Save macro name 146 ⟩ ≡

```
{
  char *s = name->spelling;
  int len = strlen(s) - 1;
  push(nw_char, &writer);
  push('<', &writer);
  while (len > 0) {
    push(*s++, &writer);
    len--;
  }
  if (*s == ' ')
    pushs("...", &writer);
  else
    push(*s, &writer);
}◇
```

Macro referenced in 145.

⟨ Add current scrap to `name`'s uses 147 ⟩ ≡

```
{
  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;
  }
}◇
```

Macro referenced in 39, 145.

"scraps.c" 148 ≡

```
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 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

"scraps.c" 149 ≡

```
static Name *pop_scrap_name(manager, parameters)
     Manager *manager;
     Parameters *parameters;
{
  char name[100];
  char *p = name;
  int c = pop(manager);
  while (TRUE) {
    if (c == nw_char)
      ⟨ Check for end of scrap name and return 150 ⟩
    else {
      *p++ = c;
      c = pop(manager);
    }
  }
}
◇
```

File defined by 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

⟨ Check for end of scrap name and return 150 ⟩ ≡

```
{
  Name *pn;
  c = pop(manager);
  if (c == nw_char) {
    *p++ = c;
    c = pop(manager);
  }
  ⟨ Check for macro parameters 40 ⟩
  if (c == '>') {
    if (p - name > 3 && p[-1] == '.' && p[-2] == '.' && p[-3] == '.') {
      p[-3] = ' ';
      p -= 2;
    }
    *p = '\0';
    pn = prefix_add(&macro_names, name);
    return pn;
  }
  else {
    fprintf(stderr, "%s: found an internal problem (1)\n", command_name);
    exit(-1);
  }
}◇
```

Macro referenced in 149.

"scraps.c" 151 ≡

```
int write_scraps(file, defs, global_indent, indent_chars,
                    debug_flag, tab_flag, indent_flag, parameters)
     FILE *file;
     Scrap_Node *defs;
     int global_indent;
     char *indent_chars;
     char debug_flag;
     char tab_flag;
     char indent_flag;
     Parameters parameters;
{
  int indent = 0;
  while (defs) {
    ⟨ Copy defs->scrap to file 152 ⟩
    defs = defs->next;
  }
  return indent + global_indent;
}
◇
```

File defined by 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

⟨ Copy defs->scrap to file 152 ⟩ ≡

```
{
  char c;
  Manager reader;
  Parameters local_parameters = 0;
  int line_number = scrap_array(defs->scrap).file_line;
  ⟨ Insert debugging information if required 153 ⟩
  reader.scrap = scrap_array(defs->scrap).slab;
  reader.index = 0;
  c = pop(&reader);
  while (c) {
    switch (c) {
      case '\n': putc(c, file);
                 line_number++;
                 ⟨ Insert appropriate indentation 154 ⟩
                 break;
      case '\t': ⟨ Handle tab characters on output 155 ⟩
                 break;
      default:
         if (c==nw_char)
           {
             ⟨ Check for macro invocation in scrap 156 ⟩
             break;
           }
         putc(c, file);
                 indent_chars[global_indent + indent] = ' ';
                 indent++;
                 break;
    }
    c = pop(&reader);
  }
}◇
```

Macro referenced in 151.

⟨ Insert debugging information if required 153 ⟩ ≡

```
if (debug_flag) {
  fprintf(file, "\n#line %d \"%s\"\n",
          line_number, scrap_array(defs->scrap).file_name);
  ⟨ Insert appropriate indentation 154 ⟩
}◇
```

Macro referenced in 152, 156.

⟨ Insert appropriate indentation 154 ⟩ ≡

```
{
  if (indent_flag) {
    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);
  }
  indent = 0;
}◇
```

Macro referenced in 152, 153.

⟨ Handle tab characters on output 155 ⟩ ≡

```
{
  if (tab_flag)
    ⟨ Expand tab into spaces 60 ⟩
  else {
    putc('\t', file);
    indent_chars[global_indent + indent] = '\t';
    indent++;
  }
}◇
```

Macro referenced in 152.

⟨ Check for macro invocation in scrap 156 ⟩ ≡

```
{
  c = pop(&reader);
  switch (c) {
    case '_': break;
    case '<': ⟨ Copy macro into file 157 ⟩
              ⟨ Insert debugging information if required 153 ⟩
              break;
    ⟨ Handle macro parameter substitution 38 ⟩
    default:
          if(c==nw_char)
            {
              putc(c, file);
              indent_chars[global_indent + indent] = ' ';
              indent++;
              break;
            }
          /* ignore, since we should already have a warning */
              break;
  }
}◇
```

Macro referenced in 152.

⟨ Copy macro into `file` 157 ⟩ ≡

```
{
  Name *name = pop_scrap_name(&reader, &local_parameters);
  if (name->mark) {
    fprintf(stderr, "%s: recursive macro discovered involving <%s>\n",
            command_name, name->spelling);
    exit(-1);
  }
  if (name->defs) {
    name->mark = TRUE;
    indent = write_scraps(file, name->defs, global_indent + indent,
                          indent_chars, debug_flag, tab_flag, indent_flag,
                          local_parameters);
    indent -= global_indent;
    name->mark = FALSE;
  }
  else if (!tex_flag)
    fprintf(stderr, "%s: macro never defined <%s>\n",
            command_name, name->spelling);
}◇
```

Macro referenced in 156.

3.2.1 Collecting Page Numbers

⟨ Function prototypes 158 ⟩ ≡

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

Macro defined by 29, 43, 59, 77, 110, 115, 130, 158, 165, 189, 197.
Macro referenced in 1.

"scraps.c" 159 ≡

```
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 134 ⟩
        }
      }
      fclose(aux_file);
      ⟨ Add letters to scraps with duplicate page numbers 160 ⟩
    }
  }
}
◇
```

File defined by 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

⟨ Add letters to scraps with duplicate page numbers 160 ⟩ ≡

```
{
  int scrap;
  for (scrap=2; scrap<scraps; scrap++) {
    if (scrap_array(scrap-1).page == scrap_array(scrap).page) {
      if (!scrap_array(scrap-1).letter)
        scrap_array(scrap-1).letter = 'a';
      scrap_array(scrap).letter = scrap_array(scrap-1).letter + 1;
    }
  }
}◇
```

Macro referenced in 159.

3.3 Names

⟨ Type declarations 161 ⟩ ≡

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

Macro defined by 3, 161, 162.
Macro referenced in 1.

⟨ Type declarations 162 ⟩ ≡

```
typedef struct name {
  char *spelling;
  struct name *llink;
  struct name *rlink;
  Scrap_Node *defs;
  Scrap_Node *uses;
  int mark;
  char tab_flag;
  char indent_flag;
  char debug_flag;
} Name;
```

◇

Macro defined by 3, 161, 162.
Macro referenced in 1.

⟨ Global variable declarations 163 ⟩ ≡

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

◇

Macro defined by 16, 18, 20, 116, 135, 163.
Macro referenced in 1.

⟨ Global variable definitions 164 ⟩ ≡

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

◇

Macro defined by 17, 19, 21, 117, 136, 164.
Macro referenced in 13.

⟨ Function prototypes 165 ⟩ ≡

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

◇

Macro defined by 29, 43, 59, 77, 110, 115, 130, 158, 165, 189, 197.
Macro referenced in 1.

"names.c" 166 ≡

```
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 11, 166, 167, 168, 170, 171, 172, 174, 176, 180, 182, 183.

"names.c" 167 ≡

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

File defined by 11, 166, 167, 168, 170, 171, 172, 174, 176, 180, 182, 183.

"names.c" 168 ≡

```
static int ambiguous_prefix();

Name *prefix_add(root, spelling)
     Name **root;
     char *spelling;
{
  Name *node = *root;
  while (node) {
    switch (compare(node->spelling, spelling)) {
    case GREATER:   root = &node->rlink;
                    break;
    case LESS:      root = &node->llink;
                    break;
    case EQUAL:     return node;
    case EXTENSION: node->spelling = save_string(spelling);
                    return node;
    case PREFIX:    ⟨ Check for ambiguous prefix 169 ⟩
                    return node;
    }
    node = *root;
  }
  ⟨ Create new name entry 173 ⟩
}
◇
```

File defined by 11, 166, 167, 168, 170, 171, 172, 174, 176, 180, 182, 183.

Since a very short prefix might match more than one macro 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 169 ⟩ ≡

```
{
  if (ambiguous_prefix(node->llink, spelling) ||
      ambiguous_prefix(node->rlink, spelling))
    fprintf(stderr,
            "%s: ambiguous prefix @<%s...@> (%s, line %d)\n",
            command_name, spelling, source_name, source_line);
}◇
```

Macro referenced in 168.

"names.c" 170 ≡

```
static int ambiguous_prefix(node, spelling)
     Name *node;
     char *spelling;
{
  while (node) {
    switch (compare(node->spelling, spelling)) {
    case GREATER:   node = node->rlink;
                    break;
    case LESS:      node = node->llink;
                    break;
    case EQUAL:
    case EXTENSION:
    case PREFIX:    return TRUE;
    }
  }
  return FALSE;
}
```

◇

File defined by 11, 166, 167, 168, 170, 171, 172, 174, 176, 180, 182, 183.

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.

"names.c" 171 ≡

```
static int robs_strcmp(x, y)
     char *x;
     char *y;
{
  char *xx = x;
  char *yy = y;
  int xc = toupper(*xx);
  int yc = toupper(*yy);
  while (xc == yc && xc) {
    xx++;
    yy++;
    xc = toupper(*xx);
    yc = toupper(*yy);
  }
  if (xc != yc) return xc - yc;
  xc = *x;
  yc = *y;
  while (xc == yc && xc) {
    x++;
    y++;
    xc = *x;
    yc = *y;
  }
  if (isupper(xc) && islower(yc))
    return xc * 2 - (toupper(yc) * 2 + 1);
  if (islower(xc) && isupper(yc))
    return toupper(xc) * 2 + 1 - yc * 2;
  return xc - yc;
}
```

◇

File defined by 11, 166, 167, 168, 170, 171, 172, 174, 176, 180, 182, 183.

"names.c" 172 ≡

```
Name *name_add(root, spelling)
     Name **root;
     char *spelling;
{
  Name *node = *root;
  while (node) {
    int result = robs_strcmp(node->spelling, spelling);
    if (result > 0)
      root = &node->llink;
    else if (result < 0)
      root = &node->rlink;
    else
      return node;
    node = *root;
  }
  ⟨ Create new name entry 173 ⟩
}
```

◇

File defined by 11, 166, 167, 168, 170, 171, 172, 174, 176, 180, 182, 183.

⟨ Create new name entry 173 ⟩ ≡

```
{
  node = (Name *) arena_getmem(sizeof(Name));
  node->spelling = save_string(spelling);
  node->mark = FALSE;
  node->llink = NULL;
  node->rlink = NULL;
  node->uses = NULL;
  node->defs = NULL;
  node->tab_flag = TRUE;
  node->indent_flag = TRUE;
  node->debug_flag = FALSE;
  *root = node;
  return node;
}◇
```

Macro referenced in 168, 172.

Name terminated by whitespace. Also check for “per-file” flags. Keep skipping white space until we reach scrap.

"names.c" 174 ≡

```
Name *collect_file_name()
{
  Name *new_name;
  char name[100];
  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';
  new_name = name_add(&file_names, name);
  ⟨ Handle optional per-file flags 175 ⟩
  c2 = source_get();
  if (c != nw_char || (c2 != '{' && c2 != '(' && c2 != '[')) {
    fprintf(stderr, "%s: expected @{, @[, or @( after file name (%s, %d)\n",
            command_name, source_name, start_line);
    exit(-1);
  }
  return new_name;
}
◇
```

File defined by 11, 166, 167, 168, 170, 171, 172, 174, 176, 180, 182, 183.

⟨ Handle optional per-file flags 175 ⟩ ≡

```
{
  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;
          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;
  }
}◇
```

Macro referenced in 174.

Name terminated by `\n` or `@{`; but keep skipping until `@{`

"names.c" 176 ≡

```
Name *collect_macro_name()
{
  char name[100];
  char *p = name;
  int start_line = source_line;
  int c = source_get(), c2;
  while (isspace(c))
    c = source_get();
  while (c != EOF) {
    switch (c) {
      case '\t':
      case ' ':  *p++ = ' ';
                 do
                   c = source_get();
                 while (c == ' ' || c == '\t');
                 break;
      case '\n': ⟨Skip until scrap begins, then return name 179⟩
      default:
         if (c==nw_char)
           {
             ⟨Check for terminating at-sequence and return name 177⟩
             break;
           }
         *p++ = c;
                 c = source_get();
                 break;
    }
  }
  fprintf(stderr, "%s: expected macro 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 11, 166, 167, 168, 170, 171, 172, 174, 176, 180, 182, 183.

⟨Check for terminating at-sequence and return name 177⟩ ≡

```
{
  c = source_get();
  switch (c) {
    case '(':
    case '[':
    case '{': ⟨Cleanup and install name 178⟩
    default:
          if (c==nw_char)
            {
              *p++ = c;
              break;
            }
          fprintf(stderr,
                      "%s: unexpected @%c in macro definition name (%s, %d)\n",
                      command_name, c, source_name, start_line);
              exit(-1);
  }
}◇
```

Macro referenced in 176.

⟨ Cleanup and install name 178 ⟩ ≡

```
{
  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';
  return prefix_add(&macro_names, name);
}◇
```

Macro referenced in 177, 179, 181.

⟨ Skip until scrap begins, then return name 179 ⟩ ≡

```
{
  do
    c = source_get();
  while (isspace(c));
  c2 = source_get();
  if (c != nw_char || (c2 != '{' && c2 != '(' && c2 != '[')) {
    fprintf(stderr, "%s: expected @{ after macro name (%s, %d)\n",
            command_name, source_name, start_line);
    exit(-1);
  }
  ⟨ Cleanup and install name 178 ⟩
}◇
```

Macro referenced in 176.

Terminated by `@>`

"names.c" 180 ≡

```
Name *collect_scrap_name()
{
  char name[100];
  char *p = name;
  int 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 181 ⟩
             break;
           }
         if (!isgraph(c)) {
                   fprintf(stderr,
                           "%s: unexpected character in macro 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 11, 166, 167, 168, 170, 171, 172, 174, 176, 180, 182, 183.

⟨ Look for end of scrap name and return 181 ⟩ ≡

```
{
  c = source_get();
  switch (c) {

    case '(':
        scrap_name_has_parameters = 1;
        ⟨ Cleanup and install name 178 ⟩
    case '>':
        scrap_name_has_parameters = 0;
        ⟨ Cleanup and install name 178 ⟩

    default:
       if (c==nw_char)
         {
           *p++ = c;
              c = source_get();
              break;
         }
       fprintf(stderr,
                      "%s: unexpected @%c in macro invocation name (%s, %d)\n",
                      command_name, c, source_name, source_line);
              exit(-1);
  }
}◇
```

Macro referenced in 180.

"names.c" 182 ≡

```
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 11, 166, 167, 168, 170, 171, 172, 174, 176, 180, 182, 183.

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" 183 ≡

```
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 11, 166, 167, 168, 170, 171, 172, 174, 176, 180, 182, 183.

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 [1].

"scraps.c" 184 ≡

```
typedef struct name_node {
  struct name_node *next;
  Name *name;
} Name_Node;
```

◇

File defined by 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

"scraps.c" 185 ≡

```
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 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

"scraps.c" 186 ≡

```
typedef struct move_node {
  struct move_node *next;
  Goto_Node *state;
  char c;
} Move_Node;
```

◇

File defined by 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

"scraps.c" 187 ≡

```
static Goto_Node *root[128];
static int max_depth;
static Goto_Node **depths;
```

◇

File defined by 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

"scraps.c" 188 ≡

```
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 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

3.4.1 Building the Automata

⟨ Function prototypes 189 ⟩ ≡

```
extern void search();
```

◇

Macro defined by 29, 43, 59, 77, 110, 115, 130, 158, 165, 189, 197.
Macro referenced in 1.

"scraps.c" 190 ≡

```
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 193 ⟩
  ⟨ Search scraps 194 ⟩
}
```

◇

File defined by 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

"scraps.c" 191 ≡

```
static void build_gotos(tree)
     Name *tree;
{
  while (tree) {
    ⟨ Extend goto graph with tree->spelling 192 ⟩
    build_gotos(tree->rlink);
    tree = tree->llink;
  }
}
```

◇

File defined by 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

⟨ Extend goto graph with `tree->spelling` 192 ⟩ ≡

```
{
  int depth = 2;
  char *p = tree->spelling;
  char c = *p++;
  Goto_Node *q = root[c];
  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++;
  }
  q->output = (Name_Node *) arena_getmem(sizeof(Name_Node));
  q->output->next = NULL;
  q->output->name = tree;
}◇
```

Macro referenced in 191.

⟨ Build failure functions 193 ⟩ ≡

```
{
  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;
    }
  }
}◇
```

Macro referenced in 190.

3.4.2 Searching the Scraps

⟨ Search scraps 194 ⟩ ≡

```
{
  for (i=1; i<scraps; i++) {
    char c;
    Manager reader;
    Goto_Node *state = NULL;
    reader.prev = NULL;
    reader.scrap = scrap_array(i).slab;
    reader.index = 0;
    c = pop(&reader);
    while (c) {
      while (state && !goto_lookup(c, state))
        state = state->fail;
      if (state)
        state = goto_lookup(c, state);
      else
        state = root[c];
      c = pop(&reader);
      if (state && state->output) {
        Name_Node *p = state->output;
        do {
          Name *name = p->name;
          if (!reject_match(name, c, &reader) &&
              (!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;
          }
          p = p->next;
        } while (p);
      }
    }
  }
}◇
```

Macro referenced in 190.

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" 195 ≡
```
#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 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

"scraps.c" 196 ≡
```
static int reject_match(name, post, reader)
     Name *name;
     char post;
     Manager *reader;
{
  int len = strlen(name->spelling);
  char first = name->spelling[0];
  char last = name->spelling[len - 1];
  char prev = '\0';
  len = reader->index - len - 2;
  if (len >= 0)
    prev = reader->scrap->chars[len];
  else if (reader->prev)
    prev = reader->scrap->chars[SLAB_SIZE - 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;
}
◇
```
File defined by 10, 37, 127, 128, 129, 131, 132, 133, 137, 138, 139, 140, 148, 149, 151, 159, 184, 185, 186, 187, 188, 190, 191, 195, 196.

3.5 Memory Management

I manage memory using a simple scheme inspired by Hanson's idea of *arenas* [3]. 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 197 ⟩ ≡
```
extern void *arena_getmem();
extern void arena_free();
◇
```
Macro defined by 29, 43, 59, 77, 110, 115, 130, 158, 165, 189, 197.
Macro referenced in 1.

"arena.c" 198 ≡

```
typedef struct chunk {
  struct chunk *next;
  char *limit;
  char *avail;
} Chunk;
◇
```

File defined by 12, 198, 199, 200, 203.

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" 199 ≡

```
static Chunk first = { NULL, NULL, NULL };
static Chunk *arena = &first;
◇
```

File defined by 12, 198, 199, 200, 203.

3.5.1 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" 200 ≡

```
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 201 ⟩
}
◇
```

File defined by 12, 198, 199, 200, 203.

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 201 ⟩ ≡

```
{
  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 202 ⟩
}◇
```

Macro referenced in 200.

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 202 ⟩ ≡

```
{
  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;
}◇
```

Macro referenced in 201.

3.5.2 Freeing Memory

To free all the memory in the arena, we need only point `arena` back to the first empty chunk.

"arena.c" 203 ≡

```
void arena_free()
{
  arena = &first;
}
◇
```

File defined by 12, 198, 199, 200, 203.

Chapter 4

Man page

Here is the UNIX man page for nuweb:

"nuweb.1" 204 ≡

```
.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.
.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 macros 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 \fImacro-name scrap\fP. Where scrap is smallar than one page.
.br
@D \fImacro-name scrap\fP. Where scrap is bigger than one page.
.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
.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 macros (useful for \fBFortran\fR).
.br
\fB-t\fP option makes \fInuweb\fP
copy tabs untouched from input to output.
.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 macros.
.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
Marc Mengel.
Internet address \fBmengel@fnal.gov\fP.
◇
```

Chapter 5

Indices

Three sets of indices can be created automatically: an index of file names, an index of macro 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 Macros

⟨ Check for macro parameters 40 ⟩ Referenced in 150.
⟨ Check for terminating at-sequence and return name 177 ⟩ Referenced in 176.
⟨ Cleanup and install name 178 ⟩ Referenced in 177, 179, 181.
⟨ Collect include-file name 124 ⟩ Referenced in 123.
⟨ Collect user-specified index entries 144 ⟩ Referenced in 143.
⟨ Compare the temp file and the old file 114 ⟩ Referenced in 113.
⟨ Copy `defs->scrap` to `file` 152 ⟩ Referenced in 151.
⟨ Copy `source_file` into `html_file` 80 ⟩ Referenced in 79.
⟨ Copy `source_file` into `tex_file` 47 ⟩ Referenced in 45.
⟨ Copy macro into `file` 157 ⟩ Referenced in 156.
⟨ Create new name entry 173 ⟩ Referenced in 168, 172.
⟨ Create new scrap, managed by `writer` 141 ⟩ Referenced in 140.
⟨ Expand tab into spaces 60 ⟩ Referenced in 58, 95, 155.
⟨ Extend goto graph with `tree->spelling` 192 ⟩ Referenced in 191.
⟨ Fill in the middle of HTML scrap environment 87 ⟩ Referenced in 82, 84.
⟨ Fill in the middle of the scrap environment 52 ⟩ Referenced in 49, 50.
⟨ Find a new chunk of memory 201 ⟩ Referenced in 200.
⟨ Finish HTML scrap environment 88 ⟩ Referenced in 82, 84.
⟨ Finish the scrap environment 53 ⟩ Referenced in 49, 50.
⟨ Format HTML macro name 97 ⟩ Referenced in 96.
⟨ Format HTML macro parameters 42 ⟩ Referenced in 97.
⟨ Format a user HTML index entry 109 ⟩ Referenced in 108.
⟨ Format a user index entry 76 ⟩ Referenced in 75.
⟨ Format an HTML index entry 103 ⟩ Referenced in 102.
⟨ Format an index entry 70 ⟩ Referenced in 69.
⟨ Format macro name 65 ⟩ Referenced in 61.
⟨ Format macro parameters 41 ⟩ Referenced in 65.
⟨ Function prototypes 29, 43, 59, 77, 110, 115, 130, 158, 165, 189, 197 ⟩ Referenced in 1.
⟨ Global variable declarations 16, 18, 20, 116, 135, 163 ⟩ Referenced in 1.
⟨ Global variable definitions 17, 19, 21, 117, 136, 164 ⟩ Referenced in 13.
⟨ Handle `EOF` 125 ⟩ Referenced in 120.
⟨ Handle an "at" character 122 ⟩ Referenced in 120.
⟨ Handle at-sign during scrap accumulation 143 ⟩ Referenced in 142.
⟨ Handle macro invocation in scrap 145 ⟩ Referenced in 143.
⟨ Handle macro parameter substitution 38 ⟩ Referenced in 156.
⟨ Handle optional per-file flags 175 ⟩ Referenced in 174.
⟨ Handle tab characters on output 155 ⟩ Referenced in 152.
⟨ Handle the file name in `argv[arg]` 26 ⟩ Referenced in 25.
⟨ Include files 2 ⟩ Referenced in 1.
⟨ Insert appropriate indentation 154 ⟩ Referenced in 152, 153.
⟨ Insert debugging information if required 153 ⟩ Referenced in 152, 156.
⟨ Interpret HTML at-sequence 81 ⟩ Referenced in 80.
⟨ Interpret at-sequence 48 ⟩ Referenced in 47.
⟨ Interpret command-line arguments 22, 23 ⟩ Referenced in 14.
⟨ Interpret the argument string `s` 24 ⟩ Referenced in 23.
⟨ Look for end of scrap name and return 181 ⟩ Referenced in 180.
⟨ Open an include file 123 ⟩ Referenced in 122.
⟨ Operating System Dependencies 15 ⟩ Referenced in 14.
⟨ Process a file 28 ⟩ Referenced in 26.
⟨ Process the remaining arguments (file names) 25 ⟩ Referenced in 14.
⟨ Reverse cross-reference lists 36 ⟩ Referenced in 30.
⟨ Save macro name 146 ⟩ Referenced in 145.
⟨ Save macro parameters 39 ⟩ Referenced in 145.
⟨ Scan at-sequence 32 ⟩ Referenced in 31.
⟨ Scan the source file, looking for at-sequences 31 ⟩ Referenced in 30.
⟨ Search scraps 194 ⟩ Referenced in 190.
⟨ Skip commented-out code 63 ⟩ Referenced in 61, 96, 143.
⟨ Skip over index entries 62 ⟩ Referenced in 61, 96.

⟨ Skip until scrap begins, then return name 179 ⟩ Referenced in 176.
⟨ Type declarations 3, 161, 162 ⟩ Referenced in 1.
⟨ Warn (only once) about needing to rerun after Latex 134 ⟩ Referenced in 132, 159.
⟨ Write HTML abbreviated definition list 98 ⟩ Referenced in 97.
⟨ Write HTML bold tag or end 101 ⟩ Referenced in 96.
⟨ Write HTML defining scrap numbers 105 ⟩ Referenced in 103.
⟨ Write HTML file defs 89 ⟩ Referenced in 82.
⟨ Write HTML file's defining scrap numbers 104 ⟩ Referenced in 103.
⟨ Write HTML index of file names 99 ⟩ Referenced in 81.
⟨ Write HTML index of macro names 100 ⟩ Referenced in 81.
⟨ Write HTML index of user-specified names 107 ⟩ Referenced in 81.
⟨ Write HTML macro declaration 85 ⟩ Referenced in 84.
⟨ Write HTML macro definition 84 ⟩ Referenced in 81.
⟨ Write HTML macro defs 90 ⟩ Referenced in 84.
⟨ Write HTML macro refs 91 ⟩ Referenced in 84.
⟨ Write HTML output file declaration 83 ⟩ Referenced in 82.
⟨ Write HTML output file definition 82 ⟩ Referenced in 81.
⟨ Write HTML referencing scrap numbers 106 ⟩ Referenced in 103.
⟨ Write LaTeX limbo definitions 46 ⟩ Referenced in 45, 79.
⟨ Write abbreviated definition list 66 ⟩ Referenced in 65.
⟨ Write defining scrap numbers 72 ⟩ Referenced in 70.
⟨ Write file defs 54 ⟩ Referenced in 49.
⟨ Write file's defining scrap numbers 71 ⟩ Referenced in 70.
⟨ Write index of file names 67 ⟩ Referenced in 48.
⟨ Write index of macro names 68 ⟩ Referenced in 48.
⟨ Write index of user-specified names 74 ⟩ Referenced in 48.
⟨ Write macro definition 50 ⟩ Referenced in 48.
⟨ Write macro defs 55 ⟩ Referenced in 50.
⟨ Write macro refs 56 ⟩ Referenced in 50.
⟨ Write out `files->spelling` 113 ⟩ Referenced in 111.
⟨ Write output file definition 49 ⟩ Referenced in 48.
⟨ Write referencing scrap numbers 73 ⟩ Referenced in 70.

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.

Bibliography

[1] A[lfred] V. Aho and M[argaret] J. Corasick, *Efficient string matching: An aid to bibliographic search.*, CACM **18** (1975), no. 6, 333–340.

[2] Nikos Drakos, *From text to hypertext: A post-hoc rationalisation of latex2html*, Computer Networks and ISDN Systems **27** (1994), 215–224.

[3] C[hris] Hanson, *Efficient stack allocation for tail-recursive languages.*, (1990).

[4] Donald E. Knuth, *Literate programming*, The Computer Journal **27** (1984), no. 2, 97–111.

[5] ———, *The TEXbook*, Computers and Typesetting, vol. 1986aA, Addison-Wesley, Reading, MA, USA, 1986.

[6] ———, *TEX: The program*, Computers & Typesetting, vol. B, Addison-Wesley, Reading, MA, USA, 1986b1986.

[7] ———, METAFONT: THE PROGRAM, Computers & Typesetting, vol. D, Addison-Wesley, Reading, MA, USA, 1986d1986.

[8] Leslie Lamport, *LATEXa document preparation systemuser's guide and reference manual*, Addison-Wesley, Reading, MA, USA, 1985.

[9] Silvio Levy, *WEB adapted to C, another approach*, TUB **8** (1987), no. 1, 12–13.

[10] Norman Ramsey, *Literate-programming tools need not be complex*, Report at `ftp.cs.princeton.edu` in `/reports/1991/351.ps.Z`. Software at `ftp.cs.princeton.edu` in `/pub/noweb.shar.Z` and at `bellcore.com` in `/pub/norman/noweb.shar.Z`. CS-TR-351-91, Department of Computer Science, Princeton University, August 1992, Submitted to *IEEE Software*.

[11] Ross Williams, *Funnelweb user's manual*, `ftp.adelaide.edu.au` in `/pub/compression` and `/pub/funnelweb`, University of Adelaide, Adelaide, South Australia, Australia, 1992.