Nuweb
A Simple Literate Programming Tool

(Version 0.87)

Preston Briggs
preston@cs.rice.edu

[0]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 [3]. 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 [6]) 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 pretty printed, resulting in a quite impressive document.

Knuth also wrote the programs for TEX and METAFONT entirely in `WEB`, eventually publishing them in book form [5, 4]. 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 [8]. Another system, FunnelWeb, is independent of any programming language and only mildly dependent on TEX [10]. 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 [7]. 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 pretty printing Both `WEB` and `CWEB` are able to pretty print 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.

[1] There is another system similar to mine, written by Norman Ramsey, called *noweb* [9]. 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.

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 identifiers looks like in each language and we certainly don't know all the keywords. (On the other hand, see the end of Section 1.3)

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 pretty printing. Since we don't pretty print, 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 pretty printing 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.

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 to 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.

1.2 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 `.tex` 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 `.tex` 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 divided.

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.

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.

Note that macro names may be abbreviated, either during invocation or definition. For example, it would be very tedious to have to repeatedly type 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 uniquely identify the macro name, 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 in a macro name; any string of them are replaced by a single blank.

When scraps are written to an output or `.tex` 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 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 in 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.

`@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.

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 4.

Identifiers must be explicitely 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. While a file name may specify a file in another directory, the resulting `.tex` 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 `.tex` 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 two 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.

1.4 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.
- 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.

1.5 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.

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 `.tex` 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" 6a ≡
    ⟨Include files 6b⟩
    ⟨Type declarations 7a, ... ⟩
    ⟨Global variable declarations 11a, ... ⟩
    ⟨Function prototypes 16b, ... ⟩
    ◇

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

⟨Include files 6b⟩ ≡
```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
```
    ◇

Macro referenced in scrap 6a.

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 7a⟩ ≡

```
#ifndef FALSE
#define FALSE 0
#endif
#ifndef TRUE
#define TRUE (!0)
#endif
◇
```

Macro defined by scraps 7ab, 46c, 47a.
Macro referenced in scrap 6a.

⟨Type declarations 7b⟩ ≡

```
#ifdef __MWERKS__
#include "DSUtils.h"
#include <stdarg.h>
extern int error(FILE *stream, const char *format, ...);
#define warning error
#define info error
#else
#define error fprintf
#define warning fprintf
#define info fprintf
#endif
◇
```

Macro defined by scraps 7ab, 46c, 47a.
Macro referenced in scrap 6a.

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 nuweb.c will contain the driver for the whole program (see Section 2.2).

"nuweb.c" 7c ≡

```
#include "global.h"
◇
```

File defined by scraps 7c, 9e.

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" 7d ≡

```
#include "global.h"
◇
```

File defined by scraps 7d, 16c.

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" 7e ≡

```
#include "global.h"
◇
```

File defined by scraps 7e, 18e, 19a, 23b, 24a, 26b, 28b.

The code controlling the creation of the output files is in output.c (see Section 2.5).

"output.c" 8a ≡
```
#include "global.h"
```
◇

File defined by scraps 8a, 30a.

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" 8b ≡
```
#include "global.h"
```
◇

File defined by scraps 8b, 32d, 33ab, 35c.

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

"scraps.c" 9a ≡
```
#include "global.h"
```
◇

File defined by scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

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

"names.c" 9b ≡
```
#include "global.h"
```
◇

File defined by scraps 9b, 48ab, 49a, 50, 51ab, 52b, 54, 56, 57bc.

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

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

File defined by scraps 9c, 64abc, 65c.

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

"global.c" 9d ≡
```
#include "global.h"
⟨Global variable definitions 11b, ... ⟩
```
◇

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.

"nuweb.c" 9e ≡
```
#ifdef __MWERKS__
⟨Macintosh utility code 12b, ... ⟩
#else
void main(argc, argv)
     int argc;
     char **argv;
{
  int arg = 1;
  ⟨Interpret command-line arguments 11e, ... ⟩
  ⟨Process the remaining arguments (file names) 14b⟩
  exit(0);
}
#endif
```
◇

File defined by scraps 7c, 9e.

2.2.1 Command-Line Arguments

There are five possible command-line arguments:

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

`-o` Suppresses generation of the output files.

`-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).

Global flags are declared for each of the arguments.

⟨Global variable declarations 11a⟩ ≡
```
extern int tex_flag;      /* if FALSE, don't emit the .tex file */
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 */
◇
```
Macro defined by scraps 11ac, 32b, 37d, 47b.
Macro referenced in scrap 6a.

The flags are all initialized for correct default behavior.

⟨Global variable definitions 11b⟩ ≡
```
int tex_flag = TRUE;
int output_flag = TRUE;
int compare_flag = TRUE;
int verbose_flag = FALSE;
int number_flag = FALSE;
◇
```
Macro defined by scraps 11bd, 32c, 38a, 47c.
Macro referenced in scrap 9d.

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

⟨Global variable declarations 11c⟩ ≡
```
extern char *command_name;
◇
```
Macro defined by scraps 11ac, 32b, 37d, 47b.
Macro referenced in scrap 6a.

⟨Global variable definitions 11d⟩ ≡
```
char *command_name = NULL;
◇
```
Macro defined by scraps 11bd, 32c, 38a, 47c.
Macro referenced in scrap 9d.

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

⟨Interpret command-line arguments 11e⟩ ≡
```
command_name = argv[0];
◇
```
Macro defined by scraps 11ef.
Macro referenced in scrap 9e.

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

⟨Interpret command-line arguments 11f⟩ ≡
```
while (arg < argc) {
  char *s = argv[arg];
  if (*s++ == '-') {
    ⟨Interpret the argument string s 12a⟩
    arg++;
  }
  else break;
}◇
```
Macro defined by scraps 11ef.
Macro referenced in scrap 9e.

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 12a⟩ ≡
```
{
  char c = *s++;
  while (c) {
    switch (c) {
      case 'c': compare_flag = FALSE;
                break;
      case 'n': number_flag = TRUE;
                break;
      case 'o': output_flag = FALSE;
                break;
      case 't': tex_flag = FALSE;
                break;
      case 'v': verbose_flag = TRUE;
                break;
      default:  error(stderr, "%s: unexpected argument ignored.  "
                      "Usage is: %s [-cnotv] file...\n",
                      command_name, command_name);
                break;
    }
    c = *s++;
  }
}◇
```
Macro referenced in scrap 11f.

2.2.2 Macintosh Specific Code

This routine frames dialog item #1, the OK button, in the options dialog

⟨Macintosh utility code 12b⟩ ≡
```

pascal void MyItemProc(WindowPtr theDlg, short theItem)
{
  Rect   iRect;
  Handle iHndl;
  short  iType;

  GetDItem(theDlg, 1, &iType, (Handle *)&iHndl, &iRect); /* item #1 */
  PenSize(3,3);
  InsetRect(&iRect, -4, -4);
  FrameRoundRect(&iRect,16,16);
}◇
```
Macro defined by scraps 12bc, 13, 14a.
Macro referenced in scrap 9e.

If the option key is held down when the program is started we pop up a dialog box allowing the user to alter the program flags.

⟨Macintosh utility code 12c⟩ ≡
```

void CenterDialog ( short theID );

check_for_options()
{
  EventRecord theEvent;
  short itemHit;
```

```
  DialogPtr myDlg;
  ControlHandle iHndl;
  Rect iRect;
  short iType;

  GetNextEvent(everyEvent, &theEvent);
  if (theEvent.modifiers & optionKey) {
    CenterDialog(128);
    myDlg = GetNewDialog(128,0,(WindowPtr) -1);

    GetDItem(myDlg,7,&iType, (Handle *)&iHndl, &iRect);
    SetDItem(myDlg,7,iType,(Handle)MyItemProc,&iRect);
    ShowWindow(myDlg);

    do {
      ModalDialog(0, &itemHit);
      switch (itemHit){
      case 1: break;
      case 2: { GetDItem(myDlg,2,&iType,(Handle *)&iHndl,&iRect);
                tex_flag = !tex_flag; SetCtlValue(iHndl,!tex_flag); break;}
      case 3: { GetDItem(myDlg,3,&iType,(Handle *)&iHndl,&iRect);
                output_flag = !output_flag; SetCtlValue(iHndl,!output_flag);
                break;}
      case 4: { GetDItem(myDlg,4,&iType,(Handle *)&iHndl,&iRect);
                compare_flag = !compare_flag; SetCtlValue(iHndl,!compare_flag);
                break;}
      case 5: { GetDItem(myDlg,5,&iType,(Handle *)&iHndl,&iRect);
                verbose_flag = !verbose_flag; SetCtlValue(iHndl,verbose_flag);
                break;}
      case 6: { GetDItem(myDlg,6,&iType,(Handle *)&iHndl,&iRect);
                number_flag = !number_flag; SetCtlValue(iHndl,number_flag);
                break;}
    }
  } while (itemHit != ok);
  DisposDialog(myDlg); }
}◇
```

Macro defined by scraps 12bc, 13, 14a.
Macro referenced in scrap 9e.

⟨Macintosh utility code 13⟩ ≡

```
void process_file(char *filename)
{
  char source_name[256];
  char tex_name[256];
  char aux_name[256];

  char *p = filename;
  char *q = source_name;
  char *trim = q;
  char *dot = NULL;
  char c;

  command_name = "nuweb";
  c = *p++;
  while (c) {
    *q++ = c;
    if (c == '.')
      dot = q - 1;
```

```
    c = *p++;
  }
  *q = '\0';
  if (dot) {
    *dot = '\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';
  }
  ⟨Process a file 16a⟩
}◇
```

Macro defined by scraps 12bc, 13, 14a.
Macro referenced in scrap 9e.

⟨Macintosh utility code 14a⟩ ≡

```
int error(FILE *stream, const char *format, ...)
{ Str255 msg;
  int i = 0;
  va_list args;
  va_start(args, format);
  vsprintf((char *)msg, format, args);
  while (msg[i]) { if (msg[i] == '\n') msg[i] = '\r'; i++; }
  c2pstr((char *)msg);
  ParamText ( msg,  NULL, NULL, NULL );
  CenterAlert ( kAlertID );
  va_end(args);
} ◇
```

Macro defined by scraps 12bc, 13, 14a.
Macro referenced in scrap 9e.

2.2.3 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) 14b⟩ ≡

```
{
  if (arg >= argc) {
    error( stderr,  "%s: expected a file name.  "
           "Usage is: %s [-cnotv] file-name...\n", command_name, command_name);
    exit(-1);
  }
  do {
    ⟨Handle the file name in argv[arg] 15a⟩
    arg++;
  } while (arg < argc);
}◇
```

Macro referenced in scrap 9e.

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 `.tex` 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]` 15a⟩ ≡

```
{
  char source_name[256];
  char tex_name[256];
  char aux_name[256];
  ⟨Build source_name and tex_name 15b⟩
  ⟨Process a file 16a⟩
}◇
```

Macro referenced in scrap 14b.

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.

⟨Build `source_name` and `tex_name` 15b⟩ ≡

```
{
  char *p = argv[arg];
  char *q = source_name;
  char *trim = q;
  char *dot = NULL;
  char c = *p++;
  while (c) {
    *q++ = c;
    if (c == '/') {
      trim = q;
      dot = NULL;
    }
    else if (c == '.')
      dot = q - 1;
    c = *p++;
  }
  *q = '\0';
  if (dot) {
    *dot = '\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 scrap 15a.

Now that we're finally ready to process a file, it's not really too complex. We bundle most of the work into three routines `pass1` (see Section 2.3), `write_tex` (see Section 2.4), and `write_files` (see Section 2.5). After we're finished with a particular file, we must remember to release its storage (see Section 3.5).

⟨Process a file 16a⟩ ≡
```
{
  pass1(source_name);
  if (tex_flag) {
    collect_numbers(aux_name);
    write_tex(source_name, tex_name);
  }
  if (output_flag)
    write_files(file_names);
  arena_free();
}◇
```
Macro referenced in scraps 13, 15a.

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 16b⟩ ≡
```
extern void pass1();
◇
```
Macro defined by scraps 16b, 18d, 29b, 32a, 36d, 45b, 47d, 58f, 63c.
Macro referenced in scrap 6a.

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" 16c ≡
```
void pass1(file_name)
     char *file_name;
{
  if (verbose_flag)
    info(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 17a⟩
  if (tex_flag)
    search();
  ⟨Reverse cross-reference lists 18c⟩
}
◇
```
File defined by scraps 7d, 16c.

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 17a⟩ ≡
```
{
  int c = source_get();
  while (c != EOF) {
    if (c == '@')
      ⟨Scan at-sequence 17b⟩
    c = source_get();
  }
}◇
```
Macro referenced in scrap 16c.

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 17b⟩ ≡
```
{
  c = source_get();
  switch (c) {
    case 'O':
    case 'o': ⟨Build output file definition 17c⟩
              break;
    case 'D':
    case 'd': ⟨Build macro definition 18a⟩
              break;
    case '@':
    case 'u':
    case 'm':
    case 'f': /* ignore during this pass */
              break;
    default:  warning(stderr,
                      "%s: unexpected @ sequence ignored (%s, line %d)\n",
                      command_name, source_name, source_line);
              break;
  }
}◇
```
Macro referenced in scrap 17a.

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 17c⟩ ≡
```
{
  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 18b⟩
}◇
```
Macro referenced in scrap 17b.

⟨Build macro definition 18a⟩ ≡
```
{
  Name *name = collect_macro_name();
  int scrap = collect_scrap();
  ⟨Add scrap to name's definition list 18b⟩
}◇
```
Macro referenced in scrap 17b.

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 18b⟩ ≡
```
{
  Scrap_Node *def = (Scrap_Node *) arena_getmem(sizeof(Scrap_Node));
  def->scrap = scrap;
  def->next = name->defs;
  name->defs = def;
}◇
```
Macro referenced in scraps 17c, 18a.

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 18c⟩ ≡
```
{
  reverse_lists(file_names);
  reverse_lists(macro_names);
  reverse_lists(user_names);
}◇
```
Macro referenced in scrap 16c.

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 18d⟩ ≡
```
extern void write_tex();
◇
```
Macro defined by scraps 16b, 18d, 29b, 32a, 36d, 45b, 47d, 58f, 63c.
Macro referenced in scrap 6a.

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

`"latex.c"` 18e ≡
```
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 scraps 7e, 18e, 19a, 23b, 24a, 26b, 28b.

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" 19a ≡
```
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)
      info(stderr, "writing %s\n", tex_name);
    source_open(file_name);
    ⟨Copy source_file into tex_file 19b⟩
    fclose(tex_file);
  }
  else
    error(stderr, "%s: can't open %s\n", command_name, tex_name);
}
◇
```
File defined by scraps 7e, 18e, 19a, 23b, 24a, 26b, 28b.

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` 19b⟩ ≡
```
{
  int scraps = 1;
  int c = source_get();
  while (c != EOF) {
    if (c == '@')
      ⟨Interpret at-sequence 20⟩
    else {
      putc(c, tex_file);
      c = source_get();
    }
  }
}◇
```
Macro referenced in scrap 19a.

⟨Interpret at-sequence 20⟩ ≡

```
{
  int big_definition = FALSE;
  c = source_get();
  switch (c) {
    case 'O': big_definition = TRUE;
    case 'o': ⟨Write output file definition 21a⟩
              break;
    case 'D': big_definition = TRUE;
    case 'd': ⟨Write macro definition 21b⟩
              break;
    case 'f': ⟨Write index of file names 25d⟩
              break;
    case 'm': ⟨Write index of macro names 26a⟩
              break;
    case 'u': ⟨Write index of user-specified names 28a⟩
              break;
    case '@': putc(c, tex_file);
    default:  c = source_get();
              break;
  }
}◇
```

Macro referenced in scrap 19b.

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 21a⟩ ≡
```
{
  Name *name = collect_file_name();
  ⟨Begin the scrap environment 21c⟩
  fprintf(tex_file, "\\verb@\"%s\"@ {\\footnotesize ", name->spelling);
  write_single_scrap_ref(tex_file, scraps++);
  fputs(" }$\\equiv$\n", tex_file);
  ⟨Fill in the middle of the scrap environment 21d⟩
  ⟨Write file defs 22b⟩
  ⟨Finish the scrap environment 22a⟩
}◇
```
Macro referenced in scrap 20.

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 21b⟩ ≡
```
{
  Name *name = collect_macro_name();
  ⟨Begin the scrap environment 21c⟩
  fprintf(tex_file, "$\\langle$%s {\\footnotesize ", name->spelling);
  write_single_scrap_ref(tex_file, scraps++);
  fputs("}$\\rangle\\equiv$\n", tex_file);
  ⟨Fill in the middle of the scrap environment 21d⟩
  ⟨Write macro defs 22c⟩
  ⟨Write macro refs 23a⟩
  ⟨Finish the scrap environment 22a⟩
}◇
```
Macro referenced in scrap 20.

⟨Begin the scrap environment 21c⟩ ≡
```
{
  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 scraps 21ab.

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 21d⟩ ≡
```
{
  fputs("\\vspace{-1ex}\n\\begin{list}{}{} \\item\n", tex_file);
  copy_scrap(tex_file);
  fputs("$\\Diamond$\n\\end{list}\n", tex_file);
}◇
```
Macro referenced in scraps 21ab.

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 22a⟩ ≡

```
{
  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 scraps 21ab.

Formatting Cross References

⟨Write file defs 22b⟩ ≡

```
{
  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 File defined by scraps ", 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 scrap 21a.

⟨Write macro defs 22c⟩ ≡

```
{
  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 Macro defined by scraps ", tex_file);
    print_scrap_numbers(tex_file, name->defs);
  }
}◇
```

Macro referenced in scrap 21b.

⟨Write macro refs 23a⟩ ≡

```
{
  if (name->uses) {
    if (name->uses->next) {
      fputs("\\item Macro referenced in scraps ", tex_file);
      print_scrap_numbers(tex_file, name->uses);
    }
    else {
      fputs("\\item Macro referenced in scrap ", tex_file);
      write_single_scrap_ref(tex_file, name->uses->scrap);
      fputs(".\n", tex_file);
    }
  }
  else {
    fputs("\\item Macro never referenced.\n", tex_file);
    warning(stderr, "%s: <%s> never referenced.\n",
            command_name, name->spelling);
  }
  fputs("\\end{list}\n", tex_file);
}◇
```

Macro referenced in scrap 21b.

"latex.c" 23b ≡

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

File defined by scraps 7e, 18e, 19a, 23b, 24a, 26b, 28b.

Formatting a Scrap

We add a \mbox{} at the beginning of each line to avoid problems with older versions of TEX.

"latex.c" 24a ≡

```
static void copy_scrap(file)
     FILE *file;
{
  int indent = 0;
  int c = source_get();
  fputs("\\mbox{}\\verb@", file);
  while (1) {
    switch (c) {
      case '@':  ⟨Check at-sequence for end-of-scrap 24c⟩
                 break;
      case '\n': fputs("@\\\\\n\\mbox{}\\verb@", file);
                 indent = 0;
                 break;
      case '\t': ⟨Expand tab into spaces 24b⟩
                 break;
      default:   putc(c, file);
                 indent++;
                 break;
    }
    c = source_get();
  }
}
◇
```

File defined by scraps 7e, 18e, 19a, 23b, 24a, 26b, 28b.

⟨Expand tab into spaces 24b⟩ ≡

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

Macro referenced in scraps 24a, 44c.

⟨Check at-sequence for end-of-scrap 24c⟩ ≡

```
{
  c = source_get();
  switch (c) {
    case '@': fputs("@{\\tt @}\\verb@", file);
              break;
    case '|': ⟨Skip over index entries 25a⟩
    case '}': putc('@', file);
              return;
    case '<': ⟨Format macro name 25b⟩
              break;
    default:  /* ignore these since pass1 will have warned about them */
              break;
  }
}◇
```

Macro referenced in scrap 24a.

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 25a⟩ ≡
```
{
  do {
    do
      c = source_get();
    while (c != '@');
    c = source_get();
  } while (c != '}');
}◇
```
Macro referenced in scrap 24c.

⟨Format macro name 25b⟩ ≡
```
{
  Name *name = collect_scrap_name();
  fprintf(file, "@$\\langle$%s {\\footnotesize ", name->spelling);
  if (name->defs)
    ⟨Write abbreviated definition list 25c⟩
  else {
    putc('?', file);
    warning(stderr, "%s: scrap never defined <%s>\n",
            command_name, name->spelling);
  }
  fputs("}$\\rangle$\\verb@", file);
}◇
```
Macro referenced in scrap 24c.

⟨Write abbreviated definition list 25c⟩ ≡
```
{
  Scrap_Node *p = name->defs;
  write_single_scrap_ref(file, p->scrap);
  p = p->next;
  if (p)
    fputs(", \\ldots\\ ", file);
}◇
```
Macro referenced in scrap 25b.

2.4.2 Generating the Indices

⟨Write index of file names 25d⟩ ≡
```
{
  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 scrap 20.

⟨Write index of macro names 26a⟩ ≡

```
{
  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 scrap 20.

"latex.c" 26b ≡

```
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 26c⟩
    name = name->rlink;
  }
}
◇
```

File defined by scraps 7e, 18e, 19a, 23b, 24a, 26b, 28b.

⟨Format an index entry 26c⟩ ≡

```
{
  fputs("\\item ", tex_file);
  if (file_flag) {
    fprintf(tex_file, "\\verb@\"%s\"@ ", name->spelling);
    ⟨Write file's defining scrap numbers 27a⟩
  }
  else {
    fprintf(tex_file, "$\\langle$%s {\\footnotesize ", name->spelling);
    ⟨Write defining scrap numbers 27b⟩
    fputs("}$\\rangle$ ", tex_file);
    ⟨Write referencing scrap numbers 27c⟩
  }
  putc('\n', tex_file);
}◇
```

Macro referenced in scrap 26b.

⟨Write file's defining scrap numbers 27a⟩ ≡

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

Macro referenced in scrap 26c.

⟨Write defining scrap numbers 27b⟩ ≡

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

Macro referenced in scrap 26c.

⟨Write referencing scrap numbers 27c⟩ ≡

```
{
  Scrap_Node *p = name->uses;
  fputs("{\\footnotesize ", tex_file);
  if (p) {
    fputs("Referenced in scrap", tex_file);
    if (p->next) {
      fputs("s ", tex_file);
      print_scrap_numbers(tex_file, p);
    }
    else {
      putc(' ', tex_file);
      write_single_scrap_ref(tex_file, p->scrap);
      putc('.', tex_file);
    }
  }
  else
    fputs("Not referenced.", tex_file);
  putc('}', tex_file);
}◇
```

Macro referenced in scrap 26c.

⟨Write index of user-specified names 28a⟩ ≡

```
{
  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 scrap 20.

"latex.c" 28b ≡

```
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 29a⟩
    name = name->rlink;
  }
}
◇
```

File defined by scraps 7e, 18e, 19a, 23b, 24a, 26b, 28b.

⟨Format a user index entry 29a⟩ ≡

```
{
  Scrap_Node *uses = name->uses;
  if (uses) {
    int page;
    Scrap_Node *defs = name->defs;
    fprintf(tex_file, "\\item \\verb@%s@: ", name->spelling);
    if (uses->scrap < defs->scrap) {
      write_scrap_ref(tex_file, uses->scrap, TRUE, &page);
      uses = uses->next;
    }
    else {
      if (defs->scrap == uses->scrap)
        uses = uses->next;
      fputs("\\underline{", tex_file);
      write_single_scrap_ref(tex_file, defs->scrap);
      putc('}', tex_file);
      page = -2;
      defs = defs->next;
    }
    while (uses || defs) {
      if (uses && (!defs || uses->scrap < defs->scrap)) {
        write_scrap_ref(tex_file, uses->scrap, FALSE, &page);
        uses = uses->next;
      }
      else {
        if (uses && defs->scrap == uses->scrap)
          uses = uses->next;
        fputs(", \\underline{", tex_file);
        write_single_scrap_ref(tex_file, defs->scrap);
        putc('}', tex_file);
        page = -2;
        defs = defs->next;
      }
    }
    fputs(".\n", tex_file);
  }
}◇
```

Macro referenced in scrap 28b.

2.5 Writing the Output Files

⟨Function prototypes 29b⟩ ≡

```
extern void write_files();
◇
```

Macro defined by scraps 16b, 18d, 29b, 32a, 36d, 45b, 47d, 58f, 63c.
Macro referenced in scrap 6a.

```
"output.c" 30a ≡
    void write_files(files)
         Name *files;
    {
      while (files) {
        write_files(files->llink);
        ⟨Write out files->spelling 30b⟩
        files = files->rlink;
      }
    }
    ◇
```
File defined by scraps 8a, 30a.

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. Perhaps it would be better to examine `files->spelling` to find any directory information.

Note the superfluous call to `remove` before `rename`. We're using it get around a bug in some implementations of `rename`.

```
⟨Write out files->spelling 30b⟩ ≡
    {
      char indent_chars[500];
      FILE *temp_file;

    #ifdef __MWERKS__
      char *temp_name = tmpnam(0);
    #else
      char *temp_name = tempnam(".", 0);
    #endif __MWERKS__

      temp_file = fopen(temp_name, "w");
      if (!temp_file) {
        error(stderr, "%s: can't create %s for a temporary file\n",
                command_name, temp_name);
        exit(-1);
      }
      if (verbose_flag)
        info(stderr, "writing %s\n", files->spelling);
      write_scraps(temp_file, files->defs, 0, indent_chars,
                   files->debug_flag, files->tab_flag, files->indent_flag);
      fclose(temp_file);
      if (compare_flag)
        ⟨Compare the temp file and the old file 31⟩
      else {
        remove(files->spelling);
        rename(temp_name, files->spelling);
      }
    }◇
```
Macro referenced in scrap 30a.

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

⟨Compare the temp file and the old file 31⟩ ≡

```
{
  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 scrap 30b.

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 32a⟩ ≡
```
extern void source_open(); /* pass in the name of the source file */
extern int source_get();   /* no args; returns the next char or EOF */
```
◇

Macro defined by scraps 16b, 18d, 29b, 32a, 36d, 45b, 47d, 58f, 63c.
Macro referenced in scrap 6a.

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

⟨Global variable declarations 32b⟩ ≡
```
extern char *source_name;  /* name of the current file */
extern int source_line;    /* current line in the source file */
```
◇

Macro defined by scraps 11ac, 32b, 37d, 47b.
Macro referenced in scrap 6a.

⟨Global variable definitions 32c⟩ ≡
```
char *source_name = NULL;
int source_line = 0;
```
◇

Macro defined by scraps 11bd, 32c, 38a, 47c.
Macro referenced in scrap 9d.

3.1.2 Local Declarations

"input.c" 32d ≡
```
static FILE *source_file;  /* the current input file */
static int source_peek;
static int double_at;
static int include_depth;
```
◇

File defined by scraps 8b, 32d, 33ab, 35c.

"input.c" 33a ≡

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

◇

File defined by scraps 8b, 32d, 33ab, 35c.

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.

"input.c" 33b ≡

```
int source_get()
{
  int c = source_peek;
  switch (c) {
    case EOF:  ⟨Handle EOF 35b⟩
               return c;
    case '@':  ⟨Handle an “at” character 34a⟩
               return c;
    case '\n': source_line++;
    default:   source_peek = getc(source_file);
               return c;
  }
}
```

◇

File defined by scraps 8b, 32d, 33ab, 35c.

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 34a⟩ ≡

```
{
  c = getc(source_file);
  if (double_at) {
    source_peek = c;
    double_at = FALSE;
    c = '@';
  }
  else
    switch (c) {
      case 'i': ⟨Open an include file 34b⟩
                break;
      case 'f': case 'm': case 'u':
      case 'd': case 'o': case 'D': case 'O':
      case '{': case '}': case '<': case '>': case '|':
                source_peek = c;
                c = '@';
                break;
      case '@': source_peek = c;
                double_at = TRUE;
                break;
      default:  error(stderr, "%s: bad @ sequence (%s, line %d)\n",
                        command_name, source_name, source_line);
                exit(-1);
    }
}◇
```

Macro referenced in scrap 33b.

⟨Open an include file 34b⟩ ≡

```
{
  char name[256];
  if (include_depth >= 10) {
    error(stderr, "%s: include nesting too deep (%s, %d)\n",
            command_name, source_name, source_line);
    exit(-1);
  }
  ⟨Collect include-file name 35a⟩
  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) {
    error(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 scrap 34a.

⟨Collect include-file name 35a⟩ ≡
```
{
    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') {
      error(stderr, "%s: unexpected characters after file name (%s, %d)\n",
              command_name, source_name, source_line);
      exit(-1);
    }
}◇
```
Macro referenced in scrap 34b.

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 35b⟩ ≡
```
{
  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 scrap 33b.

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" 35c ≡
```
void source_open(name)
     char *name;
{
  source_file = fopen(name, "r");
  if (!source_file) {
    error(stderr, "%s: couldn't open %s\n", command_name, name);
    exit(-1);
  }
  source_name = name;
  source_line = 1;
  source_peek = getc(source_file);
  double_at = FALSE;
  include_depth = 0;
}
◇
```
File defined by scraps 8b, 32d, 33ab, 35c.

3.2 Scraps

"scraps.c" 36a ≡
```
#define SLAB_SIZE 500

typedef struct slab {
  struct slab *next;
  char chars[SLAB_SIZE];
} Slab;
◇
```
File defined by scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

"scraps.c" 36b ≡
```
typedef struct {
  char *file_name;
  int file_line;
  int page;
  char letter;
  Slab *slab;
} ScrapEntry;
◇
```
File defined by scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

"scraps.c" 36c ≡
```
static ScrapEntry *SCRAP[256];

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

static int scraps;
◇
```
File defined by scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

⟨Function prototypes 36d⟩ ≡
```
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 scraps 16b, 18d, 29b, 32a, 36d, 45b, 47d, 58f, 63c.
Macro referenced in scrap 6a.

"scraps.c" 36e ≡
```
void init_scraps()
{
  scraps = 1;
  SCRAP[0] = (ScrapEntry *) arena_getmem(256 * sizeof(ScrapEntry));
}
◇
```
File defined by scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

"scraps.c" 37a ≡
```
void write_scrap_ref(file, num, first, page)
     FILE *file;
     int num;
     int first;
     int *page;
{
  if (scrap_array(num).page >= 0) {
    if (first)
      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)
      putc('?', file);
    else
      fputs(", ?", file);
    ⟨Warn (only once) about needing to rerun after Latex 37c⟩
  }
  *page = scrap_array(num).page;
}
◇
```
File defined by scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

"scraps.c" 37b ≡
```
void write_single_scrap_ref(file, num)
     FILE *file;
     int num;
{
  int page;
  write_scrap_ref(file, num, TRUE, &page);
}
◇
```
File defined by scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

⟨Warn (only once) about needing to rerun after Latex 37c⟩ ≡
```
{
  if (!already_warned) {
    info(stderr, "%s: you'll need to rerun nuweb after running latex\n",
            command_name);
    already_warned = TRUE;
  }
}◇
```
Macro referenced in scraps 37a, 46a.

⟨Global variable declarations 37d⟩ ≡
```
extern int already_warned;
◇
```
Macro defined by scraps 11ac, 32b, 37d, 47b.
Macro referenced in scrap 6a.

⟨Global variable definitions 38a⟩ ≡

```
int already_warned = 0;
◇
```

Macro defined by scraps 11bd, 32c, 38a, 47c.
Macro referenced in scrap 9d.

"scraps.c" 38b ≡

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

File defined by scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

"scraps.c" 38c ≡

```
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 scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

"scraps.c" 38d ≡

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

File defined by scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

"scraps.c" 38e ≡

```
int collect_scrap()
{
  Manager writer;
  ⟨Create new scrap, managed by writer 39a⟩
  ⟨Accumulate scrap and return scraps++ 39b⟩
}
◇
```

File defined by scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

⟨Create new scrap, managed by `writer` 39a⟩ ≡

```
{
  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;
}◇
```

Macro referenced in scrap 38e.

⟨Accumulate scrap and return `scraps++` 39b⟩ ≡

```
{
  int c = source_get();
  while (1) {
    switch (c) {
      case EOF: error(stderr, "%s: unexpect EOF in scrap (%s, %d)\n",
                        command_name, scrap_array(scraps).file_name,
                        scrap_array(scraps).file_line);
                exit(-1);
      case '@': ⟨Handle at-sign during scrap accumulation 39c⟩
                break;
      default:  push(c, &writer);
                c = source_get();
                break;
    }
  }
}◇
```

Macro referenced in scrap 38e.

⟨Handle at-sign during scrap accumulation 39c⟩ ≡

```
{
  c = source_get();
  switch (c) {
    case '@': pushs("@@", &writer);
              c = source_get();
              break;
    case '|': ⟨Collect user-specified index entries 40a⟩
    case '}': push('\0', &writer);
              return scraps++;
    case '<': ⟨Handle macro invocation in scrap 40b⟩
              break;
    default : error(stderr, "%s: unexpected @%c in scrap (%s, %d)\n",
                      command_name, c, source_name, source_line);
              exit(-1);
  }
}◇
```

Macro referenced in scrap 39b.

⟨Collect user-specified index entries 40a⟩ ≡

```
{
  do {
    char new_name[256];
    char *p = new_name;
    int ip = 0;
    do
      c = source_get();
    while (isspace(c));
    if (c != '@') {
      Name *name;
      do {
        *p++ = c; ip++;
        c = source_get();
      } while (c != '@' && !isspace(c) && ip < 256);
      if (ip == 256) {
        error(stderr, "%s: Array bounds exceeded in scrap (%s, %d)\n",
            command_name, source_name, source_line);
        exit(-1); }
      *p = '\0';
      name = name_add(&user_names, new_name);
      if (!name->defs || name->defs->scrap != scraps) {
        Scrap_Node *def = (Scrap_Node *) arena_getmem(sizeof(Scrap_Node));
        def->scrap = scraps;
        def->next = name->defs;
        name->defs = def;
      }
    }
  } while (c != '@');
  c = source_get();
  if (c != '}') {
    error(stderr, "%s: unexpected @%c in scrap (%s, %d)\n",
            command_name, c, source_name, source_line);
    exit(-1);
  }
}◇
```

Macro referenced in scrap 39c.

⟨Handle macro invocation in scrap 40b⟩ ≡

```
{
  Name *name = collect_scrap_name();
  ⟨Save macro name 41a⟩
  ⟨Add current scrap to name's uses 41b⟩
  c = source_get();
}◇
```

Macro referenced in scrap 39c.

⟨Save macro name 41a⟩ ≡

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

Macro referenced in scrap 40b.

⟨Add current scrap to `name`'s uses 41b⟩ ≡

```
{
  if (!name->uses || name->uses->scrap != scraps) {
    Scrap_Node *use = (Scrap_Node *) arena_getmem(sizeof(Scrap_Node));
    use->scrap = scraps;
    use->next = name->uses;
    name->uses = use;
  }
}◇
```

Macro referenced in scrap 40b.

`"scraps.c"` 41c ≡

```
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 scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

"scraps.c" 42a ≡

```
static Name *pop_scrap_name(manager)
     Manager *manager;
{
  char name[256];
  char *p = name;
  int ip = 0;
  int c = pop(manager);
  while (ip < 256) {
    if (c == '@')
      ⟨Check for end of scrap name and return 42b⟩
    else {
      *p++ = c; ip++;
      c = pop(manager);
    }
  }
  error(stderr, "%s: array bounds exceeded\n", command_name);
  exit(-1);
}
◇
```

File defined by scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

⟨Check for end of scrap name and return 42b⟩ ≡

```
{
  c = pop(manager);
  if (c == '@') {
    *p++ = c; ip++;
    c = pop(manager);
  }
  else if (c == '>') {
    if (p - name > 3 && p[-1] == '.' && p[-2] == '.' && p[-3] == '.') {
      p[-3] = ' ';
      p -= 2;
    }
    *p = '\0';
    return prefix_add(&macro_names, name);
  }
  else {
    error(stderr, "%s: found an internal problem (1)\n", command_name);
    exit(-1);
  }
}◇
```

Macro referenced in scrap 42a.

"scraps.c" 43a ≡

```
int write_scraps(file, defs, global_indent, indent_chars,
                   debug_flag, tab_flag, indent_flag)
     FILE *file;
     Scrap_Node *defs;
     int global_indent;
     char *indent_chars;
     char debug_flag;
     char tab_flag;
     char indent_flag;
{
  int indent = 0;
  while (defs) {
    ⟨Copy defs->scrap to file 43b⟩
    defs = defs->next;
  }
  return indent + global_indent;
}
◇
```

File defined by scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

⟨Copy defs->scrap to file 43b⟩ ≡

```
{
  char c;
  Manager reader;
  int line_number = scrap_array(defs->scrap).file_line;
  ⟨Insert debugging information if required 44a⟩
  reader.scrap = scrap_array(defs->scrap).slab;
  reader.index = 0;
  c = pop(&reader);
  while (c) {
    switch (c) {
      case '@':  ⟨Check for macro invocation in scrap 44d⟩
                 break;
      case '\n': putc(c, file);
                 line_number++;
                 ⟨Insert appropriate indentation 44b⟩
                 break;
      case '\t': ⟨Handle tab characters on output 44c⟩
                 break;
      default:   putc(c, file);
         if ((global_indent + indent) >= 500) {
           error(stderr, "%s: indent_chars array overflow.\n",
                 command_name); exit(-1); }
         indent_chars[global_indent + indent] = ' ';
                 indent++;
                 break;
    }
    c = pop(&reader);
  }
}◇
```

Macro referenced in scrap 43a.

⟨Insert debugging information if required 44a⟩ ≡

```
if (debug_flag) {
  fprintf(file, "\n#line %d \"%s\"\n",
          line_number, scrap_array(defs->scrap).file_name);
  ⟨Insert appropriate indentation 44b⟩
}◇
```

Macro referenced in scraps 43b, 44d.

⟨Insert appropriate indentation 44b⟩ ≡

```
{
  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 scraps 43b, 44a.

⟨Handle tab characters on output 44c⟩ ≡

```
{
  if (tab_flag)
    ⟨Expand tab into spaces 24b⟩
  else {
    putc('\t', file);
    if ((global_indent + indent) >= 500) {
      error(stderr, "%s: indent_chars array overflow.\n",
            command_name); exit(-1); }
    indent_chars[global_indent + indent] = '\t';
    indent++;
  }
}◇
```

Macro referenced in scrap 43b.

⟨Check for macro invocation in scrap 44d⟩ ≡

```
{
  c = pop(&reader);
  switch (c) {
    case '@': putc(c, file);
          if ((global_indent + indent) >= 500) {
            error(stderr, "%s: indent_chars array overflow.\n",
                  command_name); exit(-1); }
              indent_chars[global_indent + indent] = ' ';
              indent++;
              break;
    case '<': ⟨Copy macro into file 45a⟩
              ⟨Insert debugging information if required 44a⟩
              break;
    default:  /* ignore, since we should already have a warning */
              break;
  }
}◇
```

Macro referenced in scrap 43b.

⟨Copy macro into `file` 45a⟩ ≡

```
{
  Name *name = pop_scrap_name(&reader);
  if (name->mark) {
    error(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);
    indent -= global_indent;
    name->mark = FALSE;
  }
  else if (!tex_flag)
    warning(stderr, "%s: macro never defined <%s>\n",
            command_name, name->spelling);
}◇
```

Macro referenced in scrap 44d.

3.2.1 Collecting Page Numbers

⟨Function prototypes 45b⟩ ≡

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

Macro defined by scraps 16b, 18d, 29b, 32a, 36d, 45b, 47d, 58f, 63c.
Macro referenced in scrap 6a.

"scraps.c" 46a ≡

```
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 37c⟩
        }
      }
      fclose(aux_file);
      ⟨Add letters to scraps with duplicate page numbers 46b⟩
    }
  }
}
◇
```

File defined by scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

⟨Add letters to scraps with duplicate page numbers 46b⟩ ≡

```
{
  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 scrap 46a.

3.3 Names

⟨Type declarations 46c⟩ ≡

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

Macro defined by scraps 7ab, 46c, 47a.
Macro referenced in scrap 6a.

⟨Type declarations 47a⟩ ≡

```
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 scraps 7ab, 46c, 47a.
Macro referenced in scrap 6a.

⟨Global variable declarations 47b⟩ ≡

```
extern Name *file_names;
extern Name *macro_names;
extern Name *user_names;
◇
```

Macro defined by scraps 11ac, 32b, 37d, 47b.
Macro referenced in scrap 6a.

⟨Global variable definitions 47c⟩ ≡

```
Name *file_names = NULL;
Name *macro_names = NULL;
Name *user_names = NULL;
◇
```

Macro defined by scraps 11bd, 32c, 38a, 47c.
Macro referenced in scrap 9d.

⟨Function prototypes 47d⟩ ≡

```
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 scraps 16b, 18d, 29b, 32a, 36d, 45b, 47d, 58f, 63c.
Macro referenced in scrap 6a.

"names.c" 48a ≡

```
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 scraps 9b, 48ab, 49a, 50, 51ab, 52b, 54, 56, 57bc.

"names.c" 48b ≡

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

◇

File defined by scraps 9b, 48ab, 49a, 50, 51ab, 52b, 54, 56, 57bc.

"names.c" 49a ≡
```
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 49b⟩
                    return node;
    }
    node = *root;
  }
  ⟨Create new name entry 52a⟩
}
◇
```
File defined by scraps 9b, 48ab, 49a, 50, 51ab, 52b, 54, 56, 57bc.

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 49b⟩ ≡
```
{
  if (ambiguous_prefix(node->llink, spelling) ||
      ambiguous_prefix(node->rlink, spelling))
    warning(stderr,
            "%s: ambiguous prefix @<%s...@> (%s, line %d)\n",
            command_name, spelling, source_name, source_line);
}◇
```
Macro referenced in scrap 49a.

"names.c" 50 ≡

```
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 scraps 9b, 48ab, 49a, 50, 51ab, 52b, 54, 56, 57bc.

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" 51a ≡

```
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 scraps 9b, 48ab, 49a, 50, 51ab, 52b, 54, 56, 57bc.

"names.c" 51b ≡

```
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 52a⟩
}
◇
```

File defined by scraps 9b, 48ab, 49a, 50, 51ab, 52b, 54, 56, 57bc.

⟨Create new name entry 52a⟩ ≡

```
{
  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 scraps 49a, 51b.

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

"names.c" 52b ≡

```
Name *collect_file_name()
{
  Name *new_name;
  char name[256];
  char *p = name;
  int ip = 0;
  int start_line = source_line;
  int c = source_get();
  while (isspace(c))
    c = source_get();
  while (isgraph(c) && ip < 256) {
    *p++ = c; ip++;
    c = source_get();
  }
  if (ip == 256) {
    error(stderr, "%s: array bounds exceeded (%s, %d)\n",
            command_name, source_name, start_line);
    exit(-1);
  }
  if (p == name) {
    error(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 53⟩
  if (c != '@' || source_get() != '{') {
    error(stderr, "%s: expected @{ after file name (%s, %d)\n",
            command_name, source_name, start_line);
    exit(-1);
  }
  return new_name;
}
◇
```

File defined by scraps 9b, 48ab, 49a, 50, 51ab, 52b, 54, 56, 57bc.

⟨Handle optional per-file flags 53⟩ ≡

```
{
  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 : warning(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 scrap 52b.

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

"names.c" 54 ≡
```
Name *collect_macro_name()
{
  char name[256];
  char *p = name;
  int ip = 0;
  int start_line = source_line;
  int c = source_get();
  while (isspace(c))
    c = source_get();
  while (c != EOF && ip < 256) {
    switch (c) {
      case '@':  ⟨Check for terminating at-sequence and return name 55a⟩
                 break;
      case '\t':
      case ' ':  *p++ = ' '; ip++;
                 do
                   c = source_get();
                 while (c == ' ' || c == '\t');
                 break;
      case '%':  *p++ = ' '; ip++;
                 do
                   c = source_get();
                 while (c != '\n');
                 do
                   c = source_get();
                 while (c == ' ' || c == '\t');
                 break;
      case '\n': ⟨Skip until scrap begins, then return name 55c⟩
      default:   *p++ = c; ip++;
                 c = source_get();
                 break;
    }
  }
  if (ip == 256)
    error(stderr, "%s: array bounds exceeded (%s, %d)\n",
          command_name, source_name, start_line);
  else
    error(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 scraps 9b, 48ab, 49a, 50, 51ab, 52b, 54, 56, 57bc.

⟨Check for terminating at-sequence and return name 55a⟩ ≡

```
{
  c = source_get();
  switch (c) {
    case '@': *p++ = c; ip++;
              break;
    case '{': ⟨Cleanup and install name 55b⟩
    default:  error(stderr,
                      "%s: unexpected @%c in macro name (%s, %d)\n",
                      command_name, c, source_name, start_line);
              exit(-1);
  }
}◇
```

Macro referenced in scrap 54.

⟨Cleanup and install name 55b⟩ ≡

```
{
  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] == ' ') {
    error(stderr, "%s: empty scrap name (%s, %d)\n",
            command_name, source_name, source_line);
    exit(-1);
  }
  *p = '\0';
  return prefix_add(&macro_names, name);
}◇
```

Macro referenced in scraps 55ac, 57a.

⟨Skip until scrap begins, then return name 55c⟩ ≡

```
{
  do
    c = source_get();
  while (isspace(c));
  if (c != '@' || source_get() != '{') {
    error(stderr, "%s: expected @{ after macro name (%s, %d)\n",
            command_name, source_name, start_line);
    exit(-1);
  }
  ⟨Cleanup and install name 55b⟩
}◇
```

Macro referenced in scrap 54.

Terminated by `@>`

"names.c" 56 ≡

```
Name *collect_scrap_name()
{
  char name[256];
  char *p = name;
  int ip = 0;
  int c = source_get();
  while (c == ' ' || c == '\t')
    c = source_get();
  while (c != EOF && ip < 256) {
    switch (c) {
      case '@':  ⟨Look for end of scrap name and return 57a⟩
                 break;
      case '\t':
      case ' ':  *p++ = ' '; ip++;
                 do
                   c = source_get();
                 while (c == ' ' || c == '\t');
                 break;
      case '%':  *p++ = ' '; ip++;
                 do
                   c = source_get();
                 while (c != '\n');
                 do
                   c = source_get();
                 while (c == ' ' || c == '\t');
                 break;
      default:   if (!isgraph(c)) {
                   error(stderr,
                           "%s: unexpected character in macro name (%s, %d)\n",
                           command_name, source_name, source_line);
                   exit(-1);
                 }
                 *p++ = c; ip++;
                 c = source_get();
                 break;
    }
  }
  if (ip == 256)
    error(stderr, "%s: array bounds exceeded (%s, %d)\n",
            command_name, source_name, source_line);
  else
    error(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 scraps 9b, 48ab, 49a, 50, 51ab, 52b, 54, 56, 57bc.

⟨Look for end of scrap name and return 57a⟩ ≡

```
{
  c = source_get();
  switch (c) {
    case '@': *p++ = c; ip++;
              c = source_get();
              break;
    case '>': ⟨Cleanup and install name 55b⟩
    default:  error(stderr,
                      "%s: unexpected @%c in macro name (%s, %d)\n",
                      command_name, c, source_name, source_line);
              exit(-1);
  }
}◇
```

Macro referenced in scrap 56.

"names.c" 57b ≡

```
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 scraps 9b, 48ab, 49a, 50, 51ab, 52b, 54, 56, 57bc.

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" 57c ≡

```
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 scraps 9b, 48ab, 49a, 50, 51ab, 52b, 54, 56, 57bc.

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 occurences 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" 58a ≡

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

File defined by scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

"scraps.c" 58b ≡

```
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 scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

"scraps.c" 58c ≡

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

File defined by scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

"scraps.c" 58d ≡

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

File defined by scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

"scraps.c" 58e ≡

```
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 scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

3.4.1 Building the Automata

⟨Function prototypes 58f⟩ ≡

```
extern void search();
◇
```

Macro defined by scraps 16b, 18d, 29b, 32a, 36d, 45b, 47d, 58f, 63c.
Macro referenced in scrap 6a.

"scraps.c" 59a ≡

```
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 61⟩
  ⟨Search scraps 62⟩
}
◇
```

File defined by scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

"scraps.c" 59b ≡

```
static void build_gotos(tree)
     Name *tree;
{
  while (tree) {
    ⟨Extend goto graph with tree->spelling 60⟩
    build_gotos(tree->rlink);
    tree = tree->llink;
  }
}
◇
```

File defined by scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

⟨Extend goto graph with `tree->spelling` 60⟩ ≡

```
{
  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 scrap 59b.

⟨Build failure functions 61⟩ ≡

```
{
  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 scrap 59a.

3.4.2 Searching the Scraps

⟨Search scraps 62⟩ ≡
```
{
  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 scrap 59a.

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 occuring 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" 63a ≡
```
#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 '<': case '>':
      return TRUE;
    default:
      return FALSE;
  }
}
◇
```
File defined by scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

"scraps.c" 63b ≡
```
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 scraps 9a, 36abce, 37ab, 38bcde, 41c, 42a, 43a, 46a, 58abcde, 59ab, 63ab.

3.5 Memory Management

I manage memory using a simple scheme inspired by Hanson's idea of *arenas* [2]. 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 63c⟩ ≡
```
extern void *arena_getmem();
extern void arena_free();
◇
```
Macro defined by scraps 16b, 18d, 29b, 32a, 36d, 45b, 47d, 58f, 63c.
Macro referenced in scrap 6a.

"arena.c" 64a ≡
```
typedef struct chunk {
  struct chunk *next;
  char *limit;
  char *avail;
} Chunk;
◇
```
File defined by scraps 9c, 64abc, 65c.

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" 64b ≡
```
static Chunk first = { NULL, NULL, NULL };
static Chunk *arena = &first;
◇
```
File defined by scraps 9c, 64abc, 65c.

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" 64c ≡
```
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 65a⟩
}
◇
```
File defined by scraps 9c, 64abc, 65c.

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 65a⟩ ≡

```
{
  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 65b⟩
}◇
```

Macro referenced in scrap 64c.

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 65b⟩ ≡

```
{
  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 scrap 65a.

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" 65c ≡

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

File defined by scraps 9c, 64abc, 65c.

Chapter 4

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.

4.1 Files

4.2 Macros

4.3 Identifiers

Knuth prints his index of indentifiers 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] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.

[2] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software – Practice and Experience*, 20(1):5–12, January 1990.

[3] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

[4] Donald E. Knuth. METAFONT: *The Program*. Computers & Typesetting. Addison-Wesley, 1986.

[5] Donald E. Knuth. *TeX: The Program*. Computers & Typesetting. Addison-Wesley, 1986.

[6] Donald E. Knuth. *The TeXbook*. Computers & Typesetting. Addison-Wesley, 1986.

[7] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, 1986.

[8] Silvio Levy and Donald E. Knuth. `CWEB` user manual: The CWEB system of structured documentation. Technical Report STAN-CS-83-977, Stanford University, October 1990. Available for anonymous ftp from `labrea.stanford.edu` in directory `pub/cweb`.

[9] Norman Ramsey. Literate-programming tools need not be complex. Submitted to IEEE Software, August 1992.

[10] Ross N. Williams. FunnelWeb user's manual, May 1992. Available for anonymous ftp from `sirius.itd.adelaide.edu.au` in directory `pub/funnelweb`.