INWEB

The Program

Chapter 2

Build 4/090319 Graham Nelson

2 Parsing a Web

`2/read`: *Reading Sections.w* To read the Contents section of the web, and through that each of the other sections in turn, and to collate all of this material into one big linear array of source-code lines.

`2/lcats`: *Line Categories.w* We are going to need to identify lines of source code as falling into 18 different categories – the start of a definition, a piece of a comment, and so on. In this section we define constants to enumerate these categories, and provide a debugging routine to show the classification we are using on the web we've just read.

`2/parse`: *The Parser.w* To work through the program read in, assigning each line its category, and noting down other useful information as we go.

`2/ident`: *Identifiers.w* To find the identifier names of functions and structures, and monitor in which sections of the program they are used; and so to police the accuracy of declarations at the head of each section.

Reading Sections 2/read

Purpose

To read the Contents section of the web, and through that each of the other sections in turn, and to collate all of this material into one big linear array of source-code lines.

Definitions

¶1. This section describes a single, one-time event which happens early in every run of `inweb`: the reading in of the entire text of the web into memory, and the building of a suitable data structure to hold all this information, neatly docketed and reflecting its three layers – chapter, section, line.

It begins by reading the contents section, which really isn't a section at all (and perhaps we shouldn't pretend that it is by the use of the `.w` file extension, but we probably want it to have the same file extension, and its syntax is chosen so that syntax-colouring for regular sections doesn't make it look odd). When the word "section" is used in the `inweb` code, it almost always means "section other than the contents".

When the reading in is complete, the following variables have their final values:

```
$no_lines = 0;                      Total lines in literate source, excluding contents
$no_sections = 0;                   Again, excluding contents: it will eventually be at least 1
$no_chapters = 0;                   Similarly, this will be at least 1
$no_tangle_targets = 0;             And again
```

¶2. Once this phase is completed, the following arrays exist. We are pretty profligate with memory, but we can afford to be. (In the Parser section, we shall build even further arrays, but those aren't discussed here.)

For each individual chapter, numbered from 0 to `$no_chapters` minus 1 in order of declaration on the contents page, we store the following:

(C1) `chapter_sigil[]` is the sigil for the chapter owning this section: that is, the textual abbreviations used to identify chapters on the command line and elsewhere: `P` for Preliminaries, `7` for Chapter 7, `C` for Appendix C. In an apparently unchaptered project, the single chapter is called Sections and has sigil `S`: every section belongs to it.

(C2) `$chapter_title[]` is the title of the chapter, exactly as it is given in the Contents: for instance, "Chapter 3: Fresh Water Fish".

(C3) `$chapter_rubric[]` is the textual description of the chapter's purpose, as given in the Contents.

(C4) `$chapter_woven_pdf_leafname[]` is a leafname like `Appendix-A.pdf`, suitable for the woven PDF version of this individual chapter.

(C5) `$chapter_tangle_target[]` is the tangle ID number for tangling, or 0 if this is not marked for independent tangling.

¶3. Second, for each section other than the Contents (numbered 0 up to `$no_sections` minus 1, and in their canonical reading order, i.e., the order in which a human reader sees them in the typeset book):

(S1) `$section_chap[]` gives the chapter number to which the section belongs, which is between 0 and `$no_chapters` minus 1. (In a project without declared chapters, this will be 0, meaning the "Sections" pseudo-chapter.)

(S2) `$section_extent[]` is the number of lines in the section (which might be one more than the number of lines in its file, if it includes a chapter heading line).

(S3) `$section_pathname_relative_to_web[]` is the pathname of the section file within its own web folder.

(S4) `$section_leafname[]` is the leafname at the end of that. Obviously, it could be derived from (S3) without too much effort, but we often want it and it costs little to store, and since neither (S3) nor (S4) ever change we may as well keep both. It is also convenient to have a cross-referencing hash which lets us invert array number (S4): `$section_number_from_leafname{}`.

(ST) `$section_tangle_target[]` is the tangle ID number for the section – generally 0 to mean that it's part of the main tangle.

¶4. As this last implies, individual tangle targets are numbered from 0 up to `$no_tangle_targets` minus 1. Other than target 0, each target contains only a single section or a single chapter.

(A1) `$tangle_target_language[]` is the language of the tangle.

¶5. Lastly, for each individual line (numbered 0 up to `$no_lines` minus 1, and in their canonical reading order):

(L1) `$line_text[]` is the text as read in.

(L2) `$line_text_raw[]` is a duplicate for now. Later, the `$line_text[]` will be altered by parsing, whereas this won't, so it gives us access to the original form of the line.

(L3) `$line_sec[]` gives the section number (an index to the section arrays above) of the originating section. For the interleaved chapter heading lines placed between automatically chapters, the section number is the one just about to start, that is, the first section of the new chapter.

(L4) `$line_source_file_line[]` gives the line number, from 1, within the section file; or, for interleaved chapter headings, is 0.

---

§**1. Reading the contents page.** We read in the contents first, since that triggers everything else, by forcing reads of each section file in turn.

At the end, we have lines numbered 0 to `$no_lines-1` read in: this is the complete literate source of the web, so that we have the equivalent in memory of one long web file. Most of the lines come straight from the source files, but a few chapter heading lines are inserted if this is a multi-chapter web.

```
sub read_literate_source {
    read_contents_page("Contents.w");
}
```

§**2.** So here goes. The contents section has a syntax quite different from all other sections, and sets out bibliographic information about the web, the sections and their organisation, and so on.

```
sub read_contents_page {
    my $pathname_of_contents = $web_setting.$_[0];
    my $cline;                                          Line read in from the contents file
    my $clc = 0;                     Line count within contents section (for use in error messages)
    my $scanning_bibliographic_block = 1;                             Top bit, or bottom bit?
    my $scanning_chapter_purpose = 0;                Reading the bit just after the new chapter?
    my $path_to_chapter_being_read = "";                Where sections in the current chapter live
    my $titling_line_to_insert = "";                               To be inserted automagically
    $bibliographic_data{"Declare Section Usage"} = "On";                            The default
    $bibliographic_data{"Strict Usage Rules"} = "Off";                              The default
    open CP, $pathname_of_contents
        or die "inweb: can't open contents section at: $pathname_of_contents\n";
    while ($cline = <CP>) {
        $cline =~ s/\s+$//; $clc++;
        if ($cline eq "") { $scanning_bibliographic_block = 0; next; }
        if ($scanning_bibliographic_block == 1) ⟨Read the bibliographic data block at the top 3⟩;
        ⟨Read the roster of sections at the bottom 7⟩;
    }
    close CP;
    if ($verbose_about_input_switch == 1) {
        print "Read contents section: '", $leafname, "' (", $cline, " lines)\n";
    }
    ⟨Check that the required bibliographic data was supplied 4⟩;
    ⟨Create the main tangle target 6⟩;
}
```

§**3.** The bibliographic data gives lines in any order specifying values of variables with fixed names; a blank line ends the block.

```
⟨Read the bibliographic data block at the top 3⟩ ≡
    if ($cline =~ m/^(.*?)\:\s*(.*?)\s*$/) {
        my $key = $1;
        my $value = $2;
        if ($key eq "License") { $key = "Licence"; }               alias US to UK spelling
        if (($key eq "Title") || ($key eq "Short Title") || ($key eq "Author") ||
            ($key eq "Purpose") || ($key eq "Licence") ||
            ($key eq "Build Number") || ($key eq "Language") ||
            ($key eq "Index Extras") || ($key eq "Index Template") ||
            ($key eq "Cover Sheet") || ($key eq "Namespaces") ||
            ($key eq "Strict Usage Rules") || ($key eq "Declare Section Usage")) {
            $bibliographic_data{$key} = $value;
            if ((($key eq "Strict Usage Rules") || ($key eq "Declare Section Usage") ||
                ($key eq "Namespaces")) &&
                ($value ne "On") && ($value ne "Off")) {
                inweb_error_at("This setting must be 'On' or 'Off'", "Contents.w", $clc);
            }
        } else { inweb_error_at("no such bibliographic datum as '$key'", "Contents.w", $clc); }
    } else { inweb_error_at("expected 'Setting: Value' but found '$cline'", "Contents.w", $clc); }
    next;
```

This code is used in §2.

§4. Some bibliographic data settings are compulsory:

⟨Check that the required bibliographic data was supplied 4⟩ ≡

```
    ensure_setting_of("Title"); ensure_setting_of("Author");
    ensure_setting_of("Purpose"); ensure_setting_of("Language");
    $bibliographic_data{"Inweb Build"} = $INWEB_BUILD;
    $bibliographic_data{"Main Language"} = $bibliographic_data{"Language"};
```

This code is used in §2.

§5. Which requires the following:

```
sub ensure_setting_of {
    my $setting = $_[0];
    if (not (exists ($bibliographic_data{$setting}))) {
        inweb_fatal_error("The Contents.w section does not specify '$setting: ...'");
    }
}
```

§6. Tangle target 0 is the main program contained in the web we're reading; the programming language used by this will the one given by the `Language:` field in the contents section.

⟨Create the main tangle target 6⟩ ≡

```
    $tangle_target_language[0] = $bibliographic_data{"Language"};
    $no_tangle_targets++;
```

This code is used in §2.

§7. In the bulk of the contents, we find indented lines for sections and unindented ones for chapters.

⟨Read the roster of sections at the bottom 7⟩ ≡

```
    $cline =~ m/^(\s*)(.*?)$/;                         A pattern which cannot fail to match
    my $whitespace = $1; local $title = $2;
    if ($whitespace eq "") {
        if ($cline =~ m/^\"(.*)$/) { $scanning_chapter_purpose = 1; $cline = $1; }
        if ($scanning_chapter_purpose == 1) ⟨Record the purpose of the current chapter 8⟩
        else ⟨Read about a new chapter 9⟩;
    } else ⟨Read about, and read in, a new section 12⟩;
    next;
```

This code is used in §2.

§8. After a declared chapter heading, subsequent lines form its purpose, until we reach a closed quote: we then stop, but remove the quotation marks. Because we like a spoonful of syntactic sugar on our porridge, that's why.

⟨Record the purpose of the current chapter 8⟩ ≡

```
    if ($cline =~ m/^(.*)\"\s*$/) { $cline = $1; $scanning_chapter_purpose = 0; }
    if ($chapter_rubric[$no_chapters-1] ne "") {
        $chapter_rubric[$no_chapters-1] .= " ";
    }
    $chapter_rubric[$no_chapters-1] .= $cline;
    next;
```

This code is used in §7.

§**9.** The title tells us everything we need to know about a chapter:

⟨Read about a new chapter 9⟩ ≡

```
    my $new_chapter_sigil = "";                                   e.g., P, 1, 2, 3, A, B, ...
    my $pdf_leafname = "";
    my $ind_target = 0;
    if ($title =~ m/^(.*?)\s*\(\s*Independent\s*(.*?)\s*\)\s*$/)
        ⟨Mark this chapter as an independent tangle target 10⟩;
    if ($title eq "Sections") {
        $new_chapter_sigil = "S"; $path_to_chapter_being_read = "Sections/";
        $titling_line_to_insert = "";
        $pdf_leafname = "Sections.pdf";
        $web_is_chaptered = 0;
    } elsif ($title eq "Preliminaries") {
        $new_chapter_sigil = "P"; $path_to_chapter_being_read = "Preliminaries/";
        $titling_line_to_insert = "";
        $pdf_leafname = "Preliminaries.pdf";
        $web_is_chaptered = 1;
    } elsif ($title =~ m/^Chapter\s+(\d+)\:\s*(.*?)$/) {
        $new_chapter_sigil = $1; $path_to_chapter_being_read = "Chapter $1/";
        $titling_line_to_insert = $title.".";
        $pdf_leafname = "Chapter-$1.pdf";
        $web_is_chaptered = 1;
    } elsif ($title =~ m/^Appendix\s+([A-O])\:\s*(.*?)$/) {
        $new_chapter_sigil = $1; $path_to_chapter_being_read = "Appendix $1/";
        $titling_line_to_insert = $title.".";
        $pdf_leafname = "Appendix-$1.pdf";
        $web_is_chaptered = 1;
    } else {
        inweb_error_at("segment '$title' not understood", "Contents.w", $clc);
        print STDERR "(Must be 'Chapter <number>: Title', 'Appendix <letter A to O>: Title',\n";
        print STDERR "'Preliminaries' or 'Sections')\n";
    }
    ⟨Create the new chapter with these details 11⟩;
```

This code is used in §7.

§**10.** A chapter whose title marks it as Independent becomes a new tangle target, with the same language as the main web unless stated otherwise.

⟨Mark this chapter as an independent tangle target 10⟩ ≡

```
    $title = $1; $lang = $2;
    $current_tangle_target = ++$no_tangle_targets;
    $ind_target = $current_tangle_target;
    $tangle_target_language[$no_tangle_targets] = $bibliographic_data{"Language"};
    if ($lang ne "") { $tangle_target_language[$no_tangle_targets] = $lang; }
```

This code is used in §9.

§11.

⟨Create the new chapter with these details 11⟩ ≡

```
    $chapter_sigil[$no_chapters] = $new_chapter_sigil;
    $chapter_title[$no_chapters] = $title;
    $chapter_rubric[$no_chapters] = "";
    $chapter_tangle_target[$no_chapters] = $ind_target;
    $chapter_woven_pdf_leafname[$no_chapters] = $pdf_leafname;
    if ($ind_target == 0) { $current_tangle_target = 0; }
    $no_chapters++;
```

This code is used in §9.

§12. That's enough on creating chapters. This is the more interesting business of registering a new section within a chapter – more interesting because we also read in and process its file.

⟨Read about, and read in, a new section 12⟩ ≡

```
    my $source_file_extension = ".w";
    if ($title =~ m/^(.*?)\s*\(\s*Independent\s*(.*?)\s*\)\s*$/) {
        ⟨Mark this section as an independent tangle target 13⟩;
    } else {
        $section_tangle_target[$no_sections] = $current_tangle_target;
    }
    $section_chap[$no_sections] = $no_chapters - 1;
    $section_extent[$no_sections] = 0;
    my $path_to_section = $path_to_chapter_being_read.$title.$source_file_extension;
    $path_to_section =~ s/ Template\.i6t$/\.i6t/;
    $section_pathname_relative_to_web[$no_sections] = $path_to_section;
    $section_leafname[$no_sections] = $path_to_section;
    if ($section_leafname[$no_sections] =~ m/\/([^\/]*?)$/) {
        $section_leafname[$no_sections] = $1;
    }
    $section_number_from_leafname{$section_leafname[$no_sections]} = $no_sections;
    read_file($path_to_section, $titling_line_to_insert, $no_sections);
    $no_sections++;
```

This code is used in §7.

§13. Just as for chapters, but a section which is an independent target with language ”Inform 6” is given the filename extension `.i6t` instead of `.w`. This is to conform with the naming convention used within Inform, where I6 template files – `inweb` files with language Inform 6 – are given the file extensions `.i6t`.

⟨Mark this section as an independent tangle target 13⟩ ≡

```
    $title = $1; $lang = $2;
    $section_tangle_target[$no_sections] = ++$no_tangle_targets;
    $tangle_target_language[$no_tangle_targets] = $bibliographic_data{"Language"};
    if ($lang ne "") {
        $tangle_target_language[$no_tangle_targets] = $lang;
        if ($lang eq "Inform 6") { $source_file_extension = ".i6t"; }
    }
```

This code is used in §12.

§14. Reading source files. Note that we assume here that trailing whitespace on a line (up to but not including the line break) is not significant in the language being tangled for.

```
sub read_file {
    my $path_relative_to_web = $_[0];
    my $titling_line_for_this_chapter = $_[1];
    my $section_number = $_[2];
    my $file_line_count = 0;
    my $pathname = $web_setting.$path_relative_to_web;
    if (($titling_line_for_this_chapter ne "") &&
        ($titling_of_current_chapter ne $titling_line_for_this_chapter)) {
        $titling_of_current_chapter = $titling_line_for_this_chapter;
        $nl = '@*** '.$titling_of_current_chapter;
        ⟨Accept this as a line belonging to this section and chapter 15⟩;
    }
    open SECTIONF, $pathname or die "inweb: Unable to open $pathname\n";
    while ($nl = <SECTIONF>) {
        $file_line_count++;
        $nl =~ s/\s+$//;                                   remove trailing whitespace and the line break
        ⟨Accept this as a line belonging to this section and chapter 15⟩;
    }
    close SECTIONF;
    if ($verbose_about_input_switch == 1) {
        print "Read section: '", $pathname, "' (", $file_line_count, " lines)\n";
    }
}
```

§15.

⟨Accept this as a line belonging to this section and chapter 15⟩ ≡

The text, with a spare copy protected from the parser's meddling:

```
    $line_text[$no_lines] = $nl;
    $line_text_raw[$no_lines] = $nl;
```

And where it occurs in the web:

```
    $line_sec[$no_lines] = $section_number;
    $line_source_file_line[$no_lines] = $file_line_count;
```

And keep count:

```
    $no_lines++;
    $section_extent[$section_number]++;                 Not the same as the file line count!
```

This code is used in §14.

§16. A utility routine we need because the titling lines are special. Lines with a count of 1 begin their sections, obviously, but there are occasional lines with a count of 0 as well: these are the inserted chapter headings, and only occur in a chaptered web.

```
sub line_is_in_heading_position {
    my $i = $_[0];
    if (($line_source_file_line[$i] == 0) ||
        ($line_source_file_line[$i] == 1)) { return 1; }
    return 0;
}
```

Line Categories 2/lcats

Purpose

We are going to need to identify lines of source code as falling into 18 different categories – the start of a definition, a piece of a comment, and so on. In this section we define constants to enumerate these categories, and provide a debugging routine to show the classification we are using on the web we've just read.

Definitions

¶1. The line categories are enumerated as follows:

define `$NO_LCAT 0` *none set as yet*
define `$COMMENT_BODY_LCAT 1`
define `$MACRO_DEFINITION_LCAT 2`
define `$BAR_LCAT 3`
define `$INDEX_ENTRY_LCAT 4`
define `$PURPOSE_LCAT 5`
define `$INTERFACE_LCAT 6`
define `$GRAMMAR_LCAT 7`
define `$DEFINITIONS_LCAT 8`
define `$PARAGRAPH_START_LCAT 9`
define `$BEGIN_VERBATIM_LCAT 10`
define `$TEXT_EXTRACT_LCAT 11`
define `$BEGIN_DEFINITION_LCAT 12`
define `$GRAMMAR_BODY_LCAT 13`
define `$INTERFACE_BODY_LCAT 14`
define `$CODE_BODY_LCAT 15`
define `$CONT_DEFINITION_LCAT 19`
define `$SOURCE_DISPLAY_LCAT 16`
define `$TOGGLE_WEAVING_LCAT 17`
define `$COMMAND_LCAT 18`

---

§**1.** The scanner is intended for debugging `inweb`, and simply shows the main result of reading in and parsing the web:

```
sub scan_line_categories {
    my $sigil = $_[0];
    my $confine_to = -1;
    my $sn;
    my $i;
    for ($sn=0; $sn<$no_sections; $sn++) {
        if ($section_sigil[$sn] eq $sigil) {
            $confine_to = $sn;
        }
    }
    for ($i=0; $i<$no_lines; $i++) {
        if (($confine_to >= 0) && ($confine_to != $line_sec[$i])) { next; }
        print sprintf("%04d  %16s  %s\n",
            $i, category_name($line_category[$i]), $line_text[$i]);
    }
}
```

§**2.** And the little routine which prints category names to `stdout`:

```
sub category_name {
    my $cat = $_[0];
    if ($cat == $COMMENT_BODY_LCAT) { return "COMMENT_BODY"; }
    elsif ($cat == $MACRO_DEFINITION_LCAT) { return "MACRO_DEFINITION"; }
    elsif ($cat == $BAR_LCAT) { return "BAR"; }
    elsif ($cat == $INDEX_ENTRY_LCAT) { return "INDEX_ENTRY"; }
    elsif ($cat == $PURPOSE_LCAT) { return "PURPOSE"; }
    elsif ($cat == $INTERFACE_LCAT) { return "INTERFACE"; }
    elsif ($cat == $GRAMMAR_LCAT) { return "GRAMMAR"; }
    elsif ($cat == $DEFINITIONS_LCAT) { return "DEFINITIONS"; }
    elsif ($cat == $PARAGRAPH_START_LCAT) { return "PARAGRAPH_START"; }
    elsif ($cat == $BEGIN_VERBATIM_LCAT) { return "BEGIN_CODE"; }
    elsif ($cat == $TEXT_EXTRACT_LCAT) { return "TEXT_EXTRACT"; }
    elsif ($cat == $BEGIN_DEFINITION_LCAT) { return "BEGIN_DEFINITION"; }
    elsif ($cat == $GRAMMAR_BODY_LCAT) { return "GRAMMAR_BODY"; }
    elsif ($cat == $INTERFACE_BODY_LCAT) { return "INTERFACE_BODY"; }
    elsif ($cat == $CODE_BODY_LCAT) { return "CODE_BODY"; }
    elsif ($cat == $SOURCE_DISPLAY_LCAT) { return "SOURCE_DISPLAY"; }
    elsif ($cat == $TOGGLE_WEAVING_LCAT) { return "TOGGLE_WEAVING"; }
    elsif ($cat == $COMMAND_LCAT) { return "COMMAND"; }
    elsif ($cat == $CONT_DEFINITION_LCAT) { return "CONT_DEFINITION"; }
    else { return "? cat $cat"; }
}
```

The Parser 2/parse

Purpose

To work through the program read in, assigning each line its category, and noting down other useful information as we go.

Definitions

¶1. Like the earlier reading-in stage, the parsing stage always happens in every run of `inweb`, and it happens once only. We use the arrays already built to represent the lines, sections and chapters, and we go on to create new ones, as follows. We know everything about chapters already. However, for the sections we add:

(S5) `$section_sigil[]`, which is the sigil such as `4/fish` identifying the section briefly, and is found on the titling line before the colon. (The hash `$sigil_section{}` provides a convenient inverse to this.) Similarly, `$section_namespace[]` is the optional namespace quoted.
(S6) `$section_toc[]`, which is the text of the brief table of contents of the section, in TEX marked-up form. (The weaver adds this as a headnote when setting the section.)
(S7) `$section_purpose[]`, the text of the purpose given by `@Purpose:`.
(S8) `$section_no_pars[]`, the number of paragraphs in the section.

¶2. And for the lines we add:

(L5) `$line_category[]`, which assigns every line one of the `*_LCAT` values (see the Line Categories section for details): when parsing is complete, no line is permitted still to have category `$NO_LCAT`.
(L6) `$line_operand[]`: the type and value of this depends on the line category, but basically it supplies some useful information about what the line constructs.
(L7) `$line_operand_2[]` is a second, even more optional operand.
(L8) `$line_paragraph_number[]` is the paragraph number within the section to which the line belongs (or 0, if it is somewhere at the top before paragraphs have begun). This doesn't entirely specify the paragraph, though, as there might be a paragraph 2 in Definitions above the bar and then another in the code stuff below: so we must also record the
(L9) `$line_paragraph_ornament[]` which is the TEX for the paragraph "ornament", a ¶ or § sign, according to whether we're above or below the bar.
(L10) `$line_starts_paragraph_on_new_page[]` which is simply a flag used with paragraph headings to cause a page throw, or not.
(L11) `$line_is_comment[]` is set if the weaver should treat the line as regular-type commentary, and clear if it should treat it as verbatim program code or other such quoted matter. (We could deduce this from the line category, but it's more convenient not to have to.)
(L12) `$line_occurs_in_CWEB_macro_definition[]` is a flag needed by the tangler.

¶3. Command-category lines set their (first) operand to one of the following:

define `$PAGEBREAK_CMD 1`
define `$BNF_GRAMMAR_CMD 2`
define `$THEMATIC_INDEX_CMD 3`
define `$FIGURE_CMD 4`
define `$INDEX_UNDER_CMD 5`

¶4. With the parsing done, we discover the remaining layer of structure within the web: the paragraphs. The following count variable exists only for the sake of the statistics line printed out at the end of `inweb`'s run: it counts the number of marked paragraphs in each section, plus a notional 1 for the header part of the section (titling, purpose, interface, grammar, etc.).

```
$no_paragraphs = 0;
```

¶5. Each paragraph has a "weight", which is a measure of its typographic prominence. The paragraph weights are:

(0) an ordinary paragraph with no subheading
(1) paragraph with bold label (operand 2: the title)
(2) paragraph with a subheading (operand 2: the title)
(3) paragraph with a chapter heading (operand 2: the title)

¶6. Some paragraphs define `CWEB` macros in angled brackets, and those need special handling. We parse the following data on them, using hashes keyed by the name of the macro without angle brackets attached:

(W1) `$cweb_macros_paragraph_number{}` is the paragraph number of the definition, as printed in small type after the name in any usage.
(W2) `$cweb_macros_start{}` is the first line number of the definition body.
(W3) `$cweb_macros_end{}` is the last.
(W4) `$cweb_macros_surplus_bit{}` is the possible little first fragment of macro definition on the same line as the declaration opens, following the equals sign. (It's poor style to write macros this way, I think, but for better compatibility with `CWEB`...)

¶7. We also look out for Backus-Naur form grammar in the program, collating it all together. (Obviously, this was written specifically for the Inform documentation, but a lot of programs have grammars of one kind or another, so it might as well be available for any project.) At this parsing stage, collation is all we do. The grammar lines are numbered 0 to `$bnf_grammar_lines` minus 1, and we have arrays:

(G1) `$bnf_grammar[]` holding the text of the line,
(G2) `$bnf_grammar_source[]` holding the line number it came from in the web.

§1. Sequence of parsing. As can be seen, we work in two passes. It could all be done in one if we needed to, but this is plenty fast enough, is tidier and makes the code simpler.

```
sub parse_literate_source {
    ⟨Initialise the new arrays created in parsing phase 2⟩;
    determine_line_categories();                                    Pass 1
    establish_canonical_section_names();                            Pass 2
    ⟨Count the number of paragraphs 3⟩;
}
```

§2. See Definitions above.

```
⟨Initialise the new arrays created in parsing phase 2⟩ ≡
    my $i;
    for ($i=0; $i<$no_lines; $i++) {
        $line_is_comment[$i] = 0;
        $line_category[$i] = $NO_LCAT;
        $line_paragraph_number[$i] = 0;
        $line_paragraph_ornament[$i] = "";
        $line_starts_paragraph_on_new_page[$i] = 0;
    }
    for ($i=0; $i<$no_sections; $i++) {
        $section_sigil[$i] = "";
        $section_toc[$i] = "";
        $section_purpose[$i] = "";
        $section_no_pars[$i] = 0;
    }
```

This code is used in §1.

§3.

```
⟨Count the number of paragraphs 3⟩ ≡
    my $i;
    $no_paragraphs = 0;
    for ($i=0; $i<$no_lines; $i++) {
        if ($line_category[$i] == $PARAGRAPH_START_LCAT) {
            $no_paragraphs++;
        }
    }
```

This code is used in §1.

§4. First parse. On the first run through, we assign a category to every line of the source, and set operands as appropriate. We also note down the `$section_purpose[]` and `$section_sigil[]` as we pass.

```
sub determine_line_categories {
    $comment_mode = 1;
    $grammar_mode = 0;
    CATEGORISATION: for ($i=0; $i<$no_lines; $i++) {
        $line_is_comment[$i] = $comment_mode;
        $line_category[$i] = $COMMENT_BODY_LCAT;                    Until set otherwise down below
        my $l = $line_text[$i];
        if (line_is_in_heading_position($i)) {
            language_set($tangle_target_language[$section_tangle_target[$line_sec[$i]]]);
            ⟨Rewrite the heading in CWEB-style paragraph notation but continue 5⟩;
        }
        if ($l =~ m/^\[\[\s*(.*?)\s*\]\]\s*$/)
            ⟨Parse the line as an Inweb command 6⟩;
        if (language_pagebreak_comment($l)) {
            $line_category[$i] = $COMMAND_LCAT;
            $line_operand[$i] = $PAGEBREAK_CMD;
            next CATEGORISATION;
        }
        if ($l =~ m/^\s*\@\<\s*(.*?)\s*\@\>\s*\=\s*(.*?)$/)
            ⟨Note that a CWEB macro is defined here 7⟩;
        if ($l =~ m/^\@(\S*)(.*?)$/) {
            my $command = $1;
            my $remainder = $2;
            my $succeeded = 0;
            ⟨Parse and deal with a structural marker 8⟩;
            if ($succeeded == 0) {
                inweb_error_at_program_line("don't understand @".$command." at:", $i);
            }
        }
        if ($grammar_mode == 1) ⟨Store this line as part of the BNF grammar 15⟩;
        if ($comment_mode == 1) ⟨This is a line destined for the commentary 13⟩;
        if ($comment_mode == 0) ⟨This is a line destined for the verbatim code 14⟩;
    }
}
```

§5. In a convenient if faintly sordid manoeuvre, we rewrite a heading line – either a Chapter heading or the titling line of a section – as if it were a `CWEB`-style paragraph break of a superior kind (`@*`, which is super, or `@**` which is even more so). This code is a residue of the time when `inweb` was essentially a reimplementation of `CWEB`.

⟨Rewrite the heading in CWEB-style paragraph notation but continue 5⟩ ≡

```
if ($l =~ m/^(([A-Za-z0-9_]+::\s*)*)(\S+\/[a-z][a-z0-9]+)\:\s+(.*)\s*$/) {
    $section_namespace[$line_sec[$i]] = $1;
    $section_sigil[$line_sec[$i]] = $3;
    $sigil_section{$1} = $line_sec[$i];
    $l = '@* '.$4;
    $section_namespace[$line_sec[$i]] =~ s/\s*//g;
} else {
    if (($l =~ m/^Chapter /) || ($l =~ m/^Appendix /)) {
        $l = '@** '.$l;
    }
}
```

This code is used in §4.

§6. Note that we report an error if the command isn't one we recognise: we don't simply ignore the squares and let it fall through into the tangler.

⟨Parse the line as an Inweb command 6⟩ ≡

```
my $comm = $1;
$line_category[$i] = $COMMAND_LCAT;
if ($comm =~ m/^(.*?)\s*\:\s*(.*)\s*$/) {
    $comm = $1;
    $line_operand_2[$i] = $2;
}
if ($comm eq "Page Break") { $line_operand[$i] = $PAGEBREAK_CMD; }
elsif ($comm eq "BNF Grammar") { $line_operand[$i] = $BNF_GRAMMAR_CMD; }
elsif ($comm eq "Thematic Index") { $line_operand[$i] = $THEMATIC_INDEX_CMD; }
elsif ($comm =~ m/^Index Under (.*)$/) {
    $thematic_indices{$1} .= $line_operand_2[$i]."...".$i."|";
    $line_operand[$i] = $INDEX_UNDER_CMD;
}
elsif ($comm eq "Figure") { $line_operand[$i] = $FIGURE_CMD; }
else { inweb_error("unknown command $l\n"); }
$line_is_comment[$i] = 1;
```

This code is used in §4.

§7. These are woven and tangled much like other comment lines, but we need to notice the macro definition, of course:

⟨Note that a CWEB macro is defined here 7⟩ ≡

```
$line_category[$i] = $MACRO_DEFINITION_LCAT;
$line_operand[$i] = $1;                                   The name of the macro
$line_operand_2[$i] = $2;                  Any beginning of its content on the same line
$comment_mode = 0;
$line_is_comment[$i] = 0;
$code_lcat_for_body = $CODE_BODY_LCAT;                   Code follows on subsequent lines
next CATEGORISATION;
```

This code is used in §4.

§**8.** A structural marker is introduced by an `@` in column 1, and is a structural division in the current section. There are a number of possibilities. One, where the line is a macro definition, we have already dealt with.

There are some old `CWEB` syntaxes surviving here: `@1` is the same as `@`, `@2` the same as `@p`, `@3` the same as `@*` which is reserved (by us, though not by `CWEB`) for section headings, `@4` the same as `@**` which is reserved (ditto) for chapter headings.

In practice `@*` and `@**` headings do begin on new pages, but that is accomplished by the TEX macros which typeset them, so we don't need to force a page break ourselves: which is why we don't set the `$line_starts_paragraph_on_new_page[]` flag for those.

```
⟨Parse and deal with a structural marker 8⟩ ≡
    if ($line_category[$i] == $MACRO_DEFINITION_LCAT) { $succeeded = 1; }
    ⟨Deal with structural markers above the bar 9⟩;
    ⟨Deal with the code and extract markers 10⟩;
    ⟨Deal with the define marker 11⟩;
    my $weight = -1;
    if ($command eq "") { $weight = 0; }
    if ($command eq "p") { $weight = 1; }
    if ($command eq "pp") { $weight = 1; $line_starts_paragraph_on_new_page[$i] = 1; }
    if ($command eq "*") { $weight = 2; }
    if ($command eq "**") { $weight = 2; }
    if ($command eq "***") { $weight = 3; }
    if ($command =~ m/\*(\d)/) { $weight = eval($1)-1; }
    if ($weight >= 0) ⟨Begin a new paragraph of this weight 12⟩;
```

This code is used in §4.

§**9.** The five markers down as far as the bar:

```
⟨Deal with structural markers above the bar 9⟩ ≡
    if ($command eq "Purpose:") {
        $line_category[$i] = $PURPOSE_LCAT;
        $line_operand[$i] = $remainder;
        $line_is_comment[$i] = 1;
        $section_purpose[$line_sec[$i]] = $remainder;
        $i2 = $i+1;
        while ($line_text[$i2] =~ m/[a-z]/) {
            $section_purpose[$line_sec[$i]] .= " ".$line_text[$i2];
            $i2++;
        }
        next CATEGORISATION;
    }
    if ($command eq "Interface:") {
        $line_category[$i] = $INTERFACE_LCAT;
        $line_is_comment[$i] = 1;
        $grammar_mode = 0;
        next CATEGORISATION;
    }
    if ($command eq "Grammar:") {
        $line_category[$i] = $GRAMMAR_LCAT;
        $line_is_comment[$i] = 1;
        $grammar_mode = 1;
        next CATEGORISATION;
    }
    if ($command eq "Definitions:") {
        $line_category[$i] = $DEFINITIONS_LCAT;
        $line_is_comment[$i] = 1;
        $grammar_mode = 0;
        next CATEGORISATION;
    }
    if ($command =~ m/\-\-\-\-+/) {
        $line_category[$i] = $BAR_LCAT;
        $line_is_comment[$i] = 1;
        $command = "";
        $grammar_mode = 0;
        next CATEGORISATION;
    }
```

This code is used in §8.

§10. These have identical behaviour except for whether or not to tangle what follows:

⟨Deal with the code and extract markers 10⟩ ≡

```
    if (($command eq "c") || ($command eq "x")) {
        $line_category[$i] = $BEGIN_VERBATIM_LCAT;
        if ($command eq "x") { $code_lcat_for_body = $TEXT_EXTRACT_LCAT; }
        else { $code_lcat_for_body = $CODE_BODY_LCAT; }
        $comment_mode = 0;
        $succeeded = 1;
        $line_text[$i] = "";
    }
```

This code is used in §8.

§11. Definitions are intended to translate to C preprocessor macros, Inform 6 `Constants`, and so on.

⟨Deal with the define marker 11⟩ ≡

```
    if ($command eq "d") {
        $line_category[$i] = $BEGIN_DEFINITION_LCAT;
        $code_lcat_for_body = $CONT_DEFINITION_LCAT;
        if ($remainder =~ m/^\s*(\S+)\s+(.+?)\s*$/) {
            $line_operand[$i] = $1;                              Name of term defined
            $line_operand_2[$i] = $2;                                           Value
        } else {
            $line_operand[$i] = $remainder;                      Name of term defined
            $line_operand_2[$i] = "";                                  No value given
        }
        $comment_mode = 0;
        $line_is_comment[$i] = 0;
        $succeeded = 1;
    }
```

This code is used in §8.

§12. The noteworthy thing here is the way we fool around with the text on the line of the paragraph opening. This is a hangover from the fact that we are using a line-based system (with significant `@`s in column 1 only) yet imitating `CWEB`, which is based on escape characters and sees only a stream where there's nothing special about line breaks. Anyway,

```
@p The chronology of French weaving. Auguste de Papillon (1734-56) soon
```

is split so that "The chronology of French weaving" is stored as operand 2 (the title) while the line is rewritten to read simply

```
Auguste de Papillon (1734-56) soon
```

so that it can go on to by woven or tangled exactly as the succeeding lines will be.

⟨Begin a new paragraph of this weight 12⟩ ≡

```
    $grammar_mode = 0;
    $comment_mode = 1;
    $line_is_comment[$i] = 1;
    $line_category[$i] = $PARAGRAPH_START_LCAT;
    $line_operand[$i] = $weight;                                    Weight
    $line_operand_2[$i] = "";                                        Title
    if (($weight > 0) && ($remainder =~ m/\s+(.*?)\.\s*(.*)$/)) {
        $line_operand_2[$i] = $1;                       Title up to the full stop
        $line_text[$i] = $2;                       And then some regular material
    } else {
        if ($remainder =~ m/\s+(.+?)$/) { $line_text[$i] = $1; }
        else { $line_text[$i] = ""; }
    }
    $succeeded = 1;
```

This code is used in §8.

§13. Any last special rules for lines. In commentary mode, we look for any fancy display requests, and also look to see if there are interface lines under an `@Interface:` heading (because that happens in commentary mode, though this isn't obvious).

⟨This is a line destined for the commentary 13⟩ ≡

```
    if ($line_text[$i] =~ m/^\>\>\s+(.*?)\s*$/) {
        $line_category[$i] = $SOURCE_DISPLAY_LCAT;
        $line_operand[$i] = $1;
    }
    if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE)) {
        if (scan_line_for_interface($i)) {
            $line_category[$i] = $INTERFACE_BODY_LCAT;
        }
    }
```

This code is used in §4.

§14. Note that in an `@d` definition, a blank line is treated as the end of the definition. (This is unnecessary for C, and is a point of difference with `CWEB`, but is needed for languages which don't allow multi-line definitions.)

⟨This is a line destined for the verbatim code 14⟩ ≡
```
    if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE)) {
        scan_line_for_interface($i);
    }
    if (($line_category[$i] != $BEGIN_DEFINITION_LCAT) &&
        ($line_category[$i] != $COMMAND_LCAT)) {
        $line_category[$i] = $code_lcat_for_body;
    }
    if (($line_category[$i] == $CONT_DEFINITION_LCAT) &&
        ($line_text[$i] =~ m/^\s*$/)) {
        $line_category[$i] = $COMMENT_BODY_LCAT;
        $code_lcat_for_body = $COMMENT_BODY_LCAT;
    }
    if (language_and_so_on($line_text[$i])) {
        $line_category[$i] = $TOGGLE_WEAVING_LCAT;
    }
    if ($web_language == $C_FOR_INFORM_LANGUAGE)
        ⟨Detect some NI-specific oddities and throw them into the BNF grammar 16⟩;
```
This code is used in §4.

§15. The BNF grammar. Material underneath the optional `@Grammar:` heading is stored away thus for the time being:

⟨Store this line as part of the BNF grammar 15⟩ ≡
```
    $line_category[$i] = $GRAMMAR_BODY_LCAT;
    $bnf_grammar[$bnf_grammar_lines] = $l;
    $bnf_grammar_source[$bnf_grammar_lines] = $i;
    $bnf_grammar_lines++;
    next CATEGORISATION;
```
This code is used in §4.

§16. It's convenient to auto-detect some grammar from specific code patterns in the source code to `NI`, though of course this mustn't be done for any other project:

⟨Detect some NI-specific oddities and throw them into the BNF grammar 16⟩ ≡
```
    if ($l =~ m/the_debugging_aspects\[\] =/) {
        $bnf_grammar_source[$bnf_grammar_lines] = $i;
        $bnf_grammar[$bnf_grammar_lines++] = "<debugging-aspect>";
        $da_baseline = $i;
    }
    if (($l =~ m/^\s+\{\s+\"(.+?)\"\,\s+\"(.*?)\"\,\s+\"(.*?)\"/)
        && ($i < $da_baseline+200)) {
        $bnf_grammar[$bnf_grammar_lines] = "    := $1";
        if ($2 ne "") { $bnf_grammar[$bnf_grammar_lines] .= " $2"; }
        if ($3 ne "") { $bnf_grammar[$bnf_grammar_lines] .= " $3"; }
        $bnf_grammar_lines++;
    }
```
This code is used in §14.

§17. Second parse. We work out the values of `$line_paragraph_number[]`, `$line_paragraph_ornament[]`, `$section_toc[]`, `$section_no_pars[]`; and we also create the hashes of details for the `CWEB` macro definitions.

```
sub establish_canonical_section_names {
    my $par_number_counter = 0;
    my $pnum = 0;
    my $ornament = "\\S";
    my $i;
    my $toc = "";
    my $toc_range_start = 0;
    my $toc_name = "";
    ⟨Begin a new section TOC 19⟩;
    for ($i=0; $i<$no_lines; $i++) {
        if (($line_category[$i] == $DEFINITIONS_LCAT) ||
            ($line_category[$i] == $PURPOSE_LCAT)) {
            $ornament = "\\P";
            $par_number_counter = 0;                    Start counting paras from 1
        }
        if ($line_category[$i] == $BAR_LCAT) {
            $ornament = "\\S";
            $par_number_counter = 0;                    Start counting paras from 1 again
            ⟨Complete any named paragraph range now half-done in the TOC 21⟩;
        }
        if ($line_category[$i] == $PARAGRAPH_START_LCAT) {
            $par_number_counter++;
            if ($line_operand[$i] == 2) {               weight 2, so a section heading
                if ($line_sec[$i] > 0)                  i.e., if this isn't the first section
                    ⟨Complete the TOC from the section just ended 22⟩;
                ⟨Begin a new section TOC 19⟩;
            }
            $section_no_pars[$line_sec[$i]]++;
            if ($line_operand[$i] == 1)                 weight 1: a named @p or @pp paragraph
                ⟨Begin a new named paragraph range in the TOC 20⟩;
        }
        $line_paragraph_ornament[$i] = $ornament;
        $line_paragraph_number[$i] = $par_number_counter;
        if ($line_category[$i] == $MACRO_DEFINITION_LCAT)
            ⟨Record details of this CWEB macro 18⟩;
        $line_csn[$i] = $section_sigil[$line_sec[$i]].
            '.$'.$line_paragraph_ornament[$i].'$'.
            $line_paragraph_number[$i];
    }
    ⟨Complete the TOC from the very last section 23⟩;
}
```

§18. We now record the four hashes of data about the macro just found:

⟨Record details of this CWEB macro 18⟩ ≡

```
my $j = $i;
my $name = $line_operand[$i];
$cweb_macros_paragraph_number{$name} = $par_number_counter;
$cweb_macros_start{$name} = $j+1;
$cweb_macros_surplus_bit{$name} = $line_operand_2[$j];
$j++;
while (($j<$no_lines) && ($line_category[$j] == $CODE_BODY_LCAT)) {
    $line_occurs_in_CWEB_macro_definition[$j] = 1;
    $j++;
}
$cweb_macros_end{$name} = $j;
```

This code is used in §17.

§19. The table of contents (TOC) for a section is abbreviated, and we generate it sequentially, with the following mildly convoluted logic. The idea is to end up with something like "¶2-5. Blue cheese; ¶6. Danish blue; ¶7-11. Gouda; §1-3. Slicing the cheese; §4. Wrapping in greaseproof paper." We start writing a TOC entry when hitting a new named paragraph, and write just the "¶7" part; only when we hit the next named paragraph, or the bar dividing the section, or the end of the section, or the end of the whole web, do we complete it by adding "-11. Gouda".

⟨Begin a new section TOC 19⟩ ≡

```
$toc = ""; $toc_range_start = 0;
```

This code is used in §17.

§20. The first of the four ways we might have to complete a part-made entry.

⟨Begin a new named paragraph range in the TOC 20⟩ ≡

```
$toc = complete_toc_chunk($toc, $toc_range_start, $par_number_counter-1, $toc_name);
$toc_range_start = $par_number_counter; $toc_name = $line_operand_2[$i];
if ($toc eq "") { $toc = $section_sigil[$line_sec[$i]]."."; }
else { $toc .= "; "; }
$toc .= '$'.$ornament.'$'.$par_number_counter;
```

This code is used in §17.

§21. This one is when we hit the bar.

⟨Complete any named paragraph range now half-done in the TOC 21⟩ ≡

```
$toc = complete_toc_chunk($toc, $toc_range_start, $par_number_counter, $toc_name);
$toc_range_start = 0;
```

This code is used in §17.

§22. And here we run into the end of section.

⟨Complete the TOC from the section just ended 22⟩ ≡

```
$toc = complete_toc_chunk($toc, $toc_range_start, $par_number_counter-1, $toc_name);
set_toc_for_section($line_sec[$i]-1, $toc);
```

This code is used in §17.

§23. And here, the end of the whole web.

⟨Complete the TOC from the very last section 23⟩ ≡

```
    $toc = complete_toc_chunk($toc, $toc_range_start, $par_number_counter, $toc_name);
    set_toc_for_section($line_sec[$no_lines-1], $toc);
```

This code is used in §17.

§24. Note the TeX tie ~ to prevent line breaks between the paragraph number and names, and that we use a number range only when there are multiple paragraphs.

```
sub complete_toc_chunk {
    my $toc = $_[0];
    my $toc_range_start = $_[1];
    my $pnum = $_[2];
    my $toc_name = $_[3];
    if ($toc_range_start != 0) {
        if ($pnum != $toc_range_start) { $toc .= "-".$pnum; }
        $toc .= "~".$toc_name;
    }
    return $toc;
}
```

§25. Once we have determined the table of contents for a section, we call the following:

```
sub set_toc_for_section {
    my $sect_no = $_[0];
    my $toc = $_[1];
    $section_toc[$sect_no] = $toc;
    return;                                        delete this line for a debugging trace, viz...
    if ($toc eq "") {
        print "Warning: ", $section_sigil[$sect_no], " has no TOC.\n";
    } else {
        print $section_sigil[$sect_no], " has TOC ", $toc, "\n";
    }
}
```

Identifiers 2/ident

Purpose

To find the identifier names of functions and structures, and monitor in which sections of the program they are used; and so to police the accuracy of declarations at the head of each section.

Definitions

¶1. In some languages, we can detect function and/or structure definitions, and also the places where they are used: we use this to provide a little bit of syntax-colouring in the weaver, and to provide little footnotes to sections explaining cross-section function calls.

We also check the section's declared `@Interface`, if the web is set up with "Strict Usage Rules" to "On" – though the default is "Off".

¶2. Yet further section arrays are created here:

(S9) `$section_errors[]` will hopefully be the empty text, but otherwise accumulates any errors we find in the section's declared interface to other sections. (In languages or projects not using interfaces, this is always empty and has no effect.)

(S10) `$section_declared_a_service[]` is set to indicate that the section's `@Interface` says it provides functions to the entire web and therefore doesn't want to have to detail all its client sections. (Again, this has no effect in a web not using interface declarations.)

(S11) `$section_declared_structure_interface[]` is a concatenation of the structure ownership part of the `@Interface` for a section, with blank lines removed.

(S12) `$section_I6_template_identifiers[]` is colon-separated list of Inform 6 routines invoked from the template interpreter, and is always blank unless the language is C-for-Inform.

(S13) `$section_declared_used_by_block[]` is a block of used-by declarations from the section's `@Interface`, but rejigged into section list form (see below).

(S14) `$section_declared_uses_block[]` is the same, but for the "uses" part of the `@Interface`.

(S15) `$section_correct_uses_block[]` is what (S13) ought to be, based on the actual function calls observed in the web source.

(S16) `$section_correct_used_by_block[]` is what (S14) ought to be, based on the actual function calls observed in the web source.

(S17) `$section_correct_structures_block[]` is what (S11) ought to be, based on the actual structure usage observed in the web source.

¶3. In order to detect function definitions and structure elements, we need to know all of the types in use, and in particular we make a hash called `$base_types{}` whose keys are the base type names available to the code in the web: for instance, `int` and `char` are valid base types, and so, for convenience's sake, is `void`. These are only base types, so types resulting from qualification or pointer constructions, like `unsigned int` or `char *`, do not count.

¶4. As we go, we also accumulate data on functions, which are stored in hashes indexed by the C identifier of the function name:

(F1) `$functions_line{}` is the line number where the function declaration begins.

(F2) `$functions_declared_scope{}` is the declared scope, if any, of the function in "row of stars" notation, that is:

- (0) `""` means the scope is section-wide (which if we are checking strictly means the function may only be used in the current section);
- (1) `"*"` is illegal and never happens;
- (2) `"**"` means the scope is chapter-wide;
- (3) `"***"` means the scope is web-wide;
- (4) `"****"` is used only in C-for-Inform mode, and means the scope extends to allowing calls from the Inform template interpreter, i.e., under the control of data files outside the web;
- (5) `"*****"` means the function is `main()`, where execution begins.

(F3) `$functions_return_type{}` is the C return type of the function, e.g., `"void"` or `"unsigned char **"`;

(F4) `$functions_arglist{}` is the argument list in the declaration of the function, e.g., `"int x, int y"`;

(F5) `$functions_usage_count{}` is the number of times the function is referred to throughout the web (including in its definition, so this is always at least 1);

(F6) `$functions_usage_section_list{}` records which other sections call the function, if any do; the value is a section list concatenating string entries in the form `:C-L`, where `C` is the chapter number and `L` is the leafname of the section;

(F7) `$functions_usage_concisely_described{}` is essentially the same data, but in the more concise form of `:S` where `S` is the section number – so, for instance, `:4:13` means it is found in sections 4 and 13;

(F8) `$functions_usage_verbosely_described{}` is a verbose form of the same thing, used in comments in the tangled code (and thus unnecessary, really, but it was useful during the debugging of `inweb`).

¶5. Where exactly are functions used? We track this with `$function_usage_hash{}`, whose keys are not function names but are pairs in the form

```
function_name+Section Name.w
```

Thus the existence of a key means that the function with this name is used in the section with this name: note that `+` is not legal in a C identifier. Again, definition counts as usage, so every function is listed in this hash under its own section name.

¶6. We also want to track structures, their members, and the usage of these. For convenience of looping over all structures, a hash `$structures{}` is created with no useful value but such that its list of keys is exactly the set of structures. (The list of keys of, say, (T1) below would do just as well, but that would make the loops look a bit misleading.)

Structures then use the following hashes, whose keys are their typedef names:

(T1) `$structure_declaration_line{}` is the line number on which the `typedef` for this structure is made.

(T2) `$structure_declaration_end{}` is the final line number of its `typedef`, which is always $\geq$ (T1).

(T3) `$structure_CREATE_in_section{}`, which is used in C-for-Inform mode only, records the second number in which the `CREATE(whatever)` macro is used for the structure `whatever`. (In NI, each individual structure is allowed to be created in only one section – a self-imposed rule to improve the encapsulation of the code.)

(T4) `$structure_owner{}` is the section number of the section deemed to be the "owner" of this structure, a concept used to help `inweb` to enforce some encapsulation of code around related data.

(T5) `$structure_usage_section_list{}` is a list of leafnames of the sections which access this structure's members, – not those which merely use pointers to them, which doesn't break any encapsulation rules.

(T6) `$structure_ownership_summary{}` is similar, but in a more compressed form and using sigils; this helps the weaver to print neat footnotes under structure definitions.

(T7) `$structure_incorporates{}` is a list of structures incorporated into this one, that is, which are used as members of the current one. (Pointers do not count.)

(T8) `$structure_declaration_tangled{}` is a flag indicating that the structure's `typedef` block has now been tangled.

¶7. And members use the following hashes, whose keys are the member names:

(M1) The hash `$member_usage_section_list{}` is a section list of each distinct section using the member, which is indexed by its name. A section list is a convenient string representation of a set of sections: for instance,

```
"-Mushrooms and Toadstools.w:-Badgers.w:"
```

Each entry in the list begins with a dash and ends with a colon, and the latter character is not permitted in a section leafname, so this is an unambiguous representation.

(M2) `$member_structures{}` is a list of all structures using members with this name, with a ':' placed after each structure name: for instance, `"parse_node:tree_fern:"`.

¶8. The following global variable keeps track of which `typedef struct` definition we are currently scanning:

```
$current_struct = "";
```

---

§**1. More detailed parsing for C-like languages.** This will be another distinct phase within `inweb`: something which happens once, towards the middle of the run, though this time it happens only for projects written in a C-like language. But first we have to step back in time a little and look at something which happened during the Parser phase of `inweb`: the discovery of the types.

§**2.** The hash `$base_types{}` is used simply as a set of keys: we initialise it to the names of some built-in types, then add more from any `typedef struct` definitions we find in the web.

```
sub create_base_types_hash {
    my $i;
    $base_types{"char"} = 1;
    $base_types{"int"} = 1;
    $base_types{"float"} = 1;
    $base_types{"void"} = 1;
    $base_types{"FILE"} = 1;
    for ($i=0; $i<$no_lines; $i++) {
        if ($line_text[$i] =~ m/^typedef\s+struct\s+(\S+)\s+/) {
            $base_types{$1} = 1;
        }
    }
}
```

§3. Parsing a section interface. Most of what we want to do is to go through the web and look to see how functions and structures are being used, as we shall see. In a web with Strict Usage Rules set to “On”, the results of that parsing have to exactly match the various declarations made in the sections. In such a web, each section has to contain an `@Interface:` block like the following: it is required to declare

(a) any functions it provides to the Inform 6 template interpreter (this is only used in C-for-Inform mode, of course);
(b) any other sections which call its functions (those it is “used by”);
(c) any other sections whose functions it calls (those it “uses”);
(d) any structures it creates, and whether they are accessed only in this section (“private”) or from other sections (in which case it goes on to say exactly which ones).

```
-- Defines {-callv:index_sounds}
-- Used by Chapter 5/Literal Productions.w
-- Used by Chapter 10/Bibliographic Data.w
-- Used by Chapter 10/Figures.w
-- Uses Chapter 1/Platform-Specific Definitions.w
-- Uses Chapter 2/Sound Durations.w
-- Uses Chapter 10/Figures.w
-- Owns struct blorb_sound (private)
```

§4. The Parser section offers us the chance to grab and parse any such line of interface specification by calling this routine on each possible line. If we recognise the line as being an interface syntax, we return 1; if not, we return 0. That whole process will be complete *before* the call to `parse_C_like_sections`, so the hashes we create here will all be in existence in time to be used by `parse_C_like_sections` and its subroutines.

The scanner is used not only to spot parts of the `@Interface`, but also to notice any typedefs of structures which fly by in the code. In those cases we return 0, though, as we don’t want to make the Parser categorise those lines as part of an interface.

```
sub scan_line_for_interface {
    my $i = $_[0];
    my $l = $line_text[$i];
    my $secname = $section_leafname[$line_sec[$i]];
    ⟨Try a structure definition part of the interface 5⟩;
    if ($web_language == $C_FOR_INFORM_LANGUAGE)
        ⟨Try an Inform template interpreter part of the interface 10⟩;
    ⟨Try a uses or used by part of the interface 6⟩;
    if ($current_struct ne "")
        ⟨Spot the declaration of any structure member in the current typedef 9⟩;
    ⟨Notice the beginning of a typedef structure definition 7⟩;
    ⟨Notice the end of a typedef structure definition 8⟩;
    return 0;
}
```

§**5.** We do little more than bottle this up for later:

⟨Try a structure definition part of the interface 5⟩ ≡

```
    if ($l =~ m/^\-\-\s+Owns struct (\S+)\s+/) {
        $section_declared_structure_interface[$line_sec[$i]] .= $l."\n";
        $structure_declared_owner{$1} = $secname;
        return 1;
    }
    if ($l =~ m/^\s+\!\- shared with/) {
        $section_declared_structure_interface[$line_sec[$i]] .= $l."\n";
        return 1;
    }
```

This code is used in §4.

§**6.**

⟨Try a uses or used by part of the interface 6⟩ ≡

```
    if ($l =~ m/^\-\-\s+Used by Chapter (\d+)\/(.*?)\s*$/) {
        $section_declared_used_by_block[$line_sec[$i]] .= ':'.$1.'-'.$2;
        return 1;
    }
    if ($l =~ m/^\-\-\s+Uses Chapter (\d+)\/(.*?)\s*$/) {
        $section_declared_uses_block[$line_sec[$i]] .= ':'.$1.'-'.$2;
        return 1;
    }
    if ($l =~ m/^\-\-\s+Service\: used widely\s*$/) {
        $section_declared_a_service[$line_sec[$i]] = 1;
        return 1;
    }
```

This code is used in §4.

§**7.** We exclude any structures whose names are made with the C preprocessor concatenation operator `##`, as anything with macros creating multiple structures is going to be waaaaay too hard for us to analyse. (The Memory section of `NI` does this.)

⟨Notice the beginning of a typedef structure definition 7⟩ ≡

```
    if ($l =~ m/typedef\s+struct\s+(\S+)/) {
        $candidate = $1;
        if (not ($candidate =~ m/\#\#/)) {
            $structure_declaration_line{$candidate} = $i;
            $current_struct = $candidate;
            $structures{$candidate} = 1;
        }
    }
```

This code is used in §4.

§8. The definition is required by us to end at its first close brace:

⟨Notice the end of a typedef structure definition 8⟩ ≡

```
    if (($current_struct ne "") && ($1 =~ m/^\}/)) {
        $structure_declaration_end{$current_struct} = $i;
        $current_struct = "";
    }
```

This code is used in §4.

§9. When one structure contains another one – as opposed to merely a pointer to another one – we need to take note, as this will affect the order in which we have to tangle their definitions.

⟨Spot the declaration of any structure member in the current typedef 9⟩ ≡

```
    if ($1 =~ m/^\s*struct\s+([A-Za-z_][A-Za-z0-9_]*)\s+(\**)(.*?)\s*\;/) {
        The member is either another structure or a pointer to one
        if ($2 eq "") {                                  An actual incorporation of another structure
            $structure_incorporates{$current_struct} .= '->'.$1;
        }
        member_detected($current_struct, $1, $2, $3, 1);
    } elsif ($1 =~ m/^\s*([A-Za-z_][A-Za-z0-9_]*)\s+()\(\*(.*?)\)\(.*\)\;/) {
        The member is a function pointer
        member_detected($current_struct, $1, $2, $3, 0);
    } elsif ($1 =~ m/^\s*([A-Za-z_][A-Za-z0-9_]*)\s+(\**)(.*?)\s*\;/) {
        The member has a base type which is not a structure
        member_detected($current_struct, $1, $2, $3, 0);
    }
```

This code is used in §4.

§10. For the format of Inform template commands, see the I6 template files. These declarations basically refer to Inform 6 identifier names used in compilation.

⟨Try an Inform template interpreter part of the interface 10⟩ ≡

```
    if ($1 =~ m/^\-\-\s+Defines \{-callv*\:(.*?)\}\s*$/) {
        $id = $1; $id =~ s/::/__/g;
        $section_I6_template_identifiers[$line_sec[$i]] .= $id.':';
        return 1;
    }
    if ($1 =~ m/^\-\-\s+Defines \{-array\:(.*?)\}\s*$/) {
        $id = $1; $id =~ s/::/__/g;
        $section_I6_template_identifiers[$line_sec[$i]] .= $id.'_array:';
        return 1;
    }
    if ($1 =~ m/^\-\-\s+Defines \{-routine\:(.*?)\}\s*$/) {
        $id = $1; $id =~ s/::/__/g;
        $section_I6_template_identifiers[$line_sec[$i]] .= $id.'_routine:';
        return 1;
    }
```

This code is used in §4.

§11. When a new member is detected, we need to register it in a dictionary of all structure members used in the web. These are defined in C with lines like

```
int one, two[10];
```

which would lead to the function call

```
member_detected("what_have_you", "int", "", "one, two[10]", 0);
```

being made. We need to uncouple a list such as `"one, two[10]"` and remove the punctuation and array extents to obtain the actual member names present: in this case `one` and `two`.

```
sub member_detected {
    my $structure = $_[0];
    my $return_type = $_[1];
    my $return_type_pointer_stars = $_[2];
    my $member_name = $_[3];
    my $vouched_for = $_[4];
    ⟨Recursively uncouple the member list 12⟩;
    $member_name =~ s/\[.*$//;
    if (exists($blacklisted_members{$member_name})) { return; }
    if ((exists $base_types{$return_type}) || ($vouched_for == 1)) {
        $member_structures{$member_name} .= $structure.":";
        $member_types{$member_name} .= " ".$return_type.$return_type_pointer_stars;
    } else {
        inweb_error("member '".$member_name."' of structure '".$structure.
            "' has unknown type '".$return_type."'");
    }
}
```

§12.

```
⟨Recursively uncouple the member list 12⟩ ≡
    if ($member_name =~ m/^\s*(.*?)\,\s*(.*?)\s*$/) {
        my $left_chunk = $1;
        my $right_chunk = $2;
        member_detected($structure, $return_type, $return_type_pointer_stars,
            $left_chunk, $vouched_for);
        member_detected($structure, $return_type, $return_type_pointer_stars,
            $right_chunk, $vouched_for);
        return;
    }
```

This code is used in §11.

§13. The parsing-C-like-sections stage. We can now get on with the main narrative of this section: what happens when `inweb` parses a section of a C-like-language web to look at its functions, its structures, and (if Strict Usage Rules is "On") also its `@Interface` declarations. The check comes in two parts, with any errors reported in a unified way at the end:

```
sub parse_C_like_sections {
    my $i;
    for ($i=0; $i<$no_sections; $i++) { $section_errors[$no_sections] = ""; }
    ⟨Stage I: police functions and section usage 14⟩;
    ⟨Stage II: police structures and member usage 15⟩;
    report_any_interface_errors_found();
}
```

§14.

```
⟨Stage I: police functions and section usage 14⟩ ≡
    find_function_definitions();
    scan_identifiers_in_source();
    find_actual_section_usage();
    check_interface_declarations_for_uses_and_used_by();
```

This code is used in §13.

§15.

```
⟨Stage II: police structures and member usage 15⟩ ≡
    find_structure_ownership();
    establish_structure_usage();
    check_interface_declarations_for_structures();
    if ($web_language == $C_FOR_INFORM_LANGUAGE) { check_uniqueness_of_structure_members(); }
```

This code is used in §13.

§16. Recognising C function definitions. A function has to be declared inside code, or inside an `@d` definition:

```
sub find_function_definitions {
    my $i;
    for ($i=0; $i<$no_lines; $i++) {
        if (($line_category[$i] == $CODE_BODY_LCAT)
            || ($line_category[$i] == $BEGIN_DEFINITION_LCAT)
            || ($line_category[$i] == $CONT_DEFINITION_LCAT))
            ⟨Look for a function definition on this line 17⟩;
    }
}
```

§17. We recognise a C function as being a line which starts with an optional comment in the form `/**/`, `/***/`, `/****/` or `/*****/`, and then (or instead) takes the form `type identifier(args...`, where type is either one of the base types found already or else a pointer to a type.

⟨Look for a function definition on this line 17⟩ ≡
```
    my $look_for_identifiers = $line_text[$i];
    my $stars_in_comment = "";
    my $type_qualifiers = "";
    if ($look_for_identifiers =~ m/^\s*\/(\*+)\/\s*(.*)$/) {
        $stars_in_comment = $1; $look_for_identifiers = $2;
    }
    if ($look_for_identifiers =~ m/^\s*(signed|unsigned|long)\s+(.*)$/) {
        $type_qualifiers = $1; $look_for_identifiers = $2;
    }
    if ($look_for_identifiers =~
        m/^\s*([A-Za-z_][A-Za-z0-9_]*)\s+(\**)(([A-Za-z_][A-Za-z0-9_]*|\:\:)+)\s*\((.*)$/) {
        my $return_type = $1;
        my $return_type_pointer_stars = $2;
        my $fname = $3;
        my $arguments = $5;
        if ((not(exists $blacklisted_functions{$fname})) &&
            ($bibliographic_data{"Namespaces"} eq "On")) {
            if ($stars_in_comment ne "") {
                inweb_error("with Namespaces on, $fname should not be marked /*...*/");
            }
            if (($fname =~ m/::/) && ($stars_in_comment eq "")) {
                $stars_in_comment = "***";
            }
        }
        $fname =~ s/::/__/g;
        if ((exists $base_types{$return_type}) &&
            (not(exists $blacklisted_functions{$fname}))) {
            ⟨Deal with type qualifiers and constructors 18⟩;
            ⟨Accept this as a new function definition 19⟩;
        }
    }
```
This code is used in §16.

§18. Combine the three parts of the return type:

⟨Deal with type qualifiers and constructors 18⟩ ≡
```
    if ($return_type_pointer_stars ne "") {
        $return_type .= " ".$return_type_pointer_stars;
    }
    if ($type_qualifiers ne "") {
        $return_type = $type_qualifiers." ".$return_type;
    }
```
This code is used in §17.

§**19.** In the following, we merge the text of up to the next 9 subsequent lines of code, in order to have enough of the function to be pretty sure that we've got the whole of its arguments: then we peel this off to obtain the argument list in the definition.

```
⟨Accept this as a new function definition 19⟩ ≡
    $functions_line{$fname} = $i;
    $functions_return_type{$fname} = $return_type;
    $functions_declared_scope{$fname} = $stars_in_comment;

    my $idash;
    for ($idash = $i+1;
        (($idash<$no_lines) && ($idash-$i<10));
        $idash++) {
        $arguments .= ' '.$line_text[$idash];
    }
    if ($arguments =~ m/^(.*?)\)\s+\{/) {
        $functions_arglist{$fname} = $1;
    } else {
        inweb_error("unable to find spec: $fname (args: '$arguments')");
    }
```

This code is used in §17.

§**20. Scanning for function calls and accesses of structure members.** In other words, here we look for where things are actually used, rather than where they are declared or defined. We skip material in the appendices, as these might be configuration files likely to cause false positives.

```
sub scan_identifiers_in_source {
    my $i;
    for ($i=0; $i<$no_lines; $i++) {
        if (($line_category[$i] == $CODE_BODY_LCAT)
            || ($line_category[$i] == $BEGIN_DEFINITION_LCAT)
            || ($line_category[$i] == $CONT_DEFINITION_LCAT)) {
            my $current_section_leafname = $section_leafname[$line_sec[$i]];
            my $current_section_number = $line_sec[$i];
            my $look_for_identifiers = $line_text[$i];

            if ($section_sigil[$line_sec[$i]] =~ m/^[A-O]/) { next; }
            ⟨Remove any C comments from the line 21⟩;

            if ($web_language == $C_FOR_INFORM_LANGUAGE)
                ⟨Detect some of NI's fruitier macros 25⟩;

            ⟨Detect use of C's dot operator for structure member access 22⟩;
            ⟨Detect use of C's arrow operator for structure member access 23⟩;

            while ($look_for_identifiers =~ m/^.*?([A-Za-z_]([A-Za-z0-9_]|::)*)(.*?)$/) {
                $identifier = $1;
                $look_for_identifiers = $3;
                $identifier =~ s/::/__/g;
                ⟨Is this identifier a function whose name we recognise? 24⟩;
            }
        }
    }
}
```

§21. This would be fooled by e.g. `a="/*"; call_me(); b="*/";`, but we'll take the risk. None of what we do in this section is mission-critical in that its failure would cause the program to mis-compile: the worst case is that we don't always print the right footnotes in the woven form, which would be sad, but not the end of the world.

⟨Remove any C comments from the line 21⟩ ≡

```
    $look_for_identifiers =~ s/\/\*(.*?)\*\///g;
```

This code is used in §20.

§22. For instance, spot `parse_node.word_ref1` as an access of member `word_ref1`.

⟨Detect use of C's dot operator for structure member access 22⟩ ≡

```
    while ($look_for_identifiers =~ m/^(.*?)\.([A-Za-z_][A-Za-z0-9_]*)(.*?)$/) {
        $look_for_identifiers = $1.$3;
        note_usage_of_structure_member($2, $i);
    }
```

This code is used in §20.

§23. And almost identically but for `->`, with pointers to structures:

⟨Detect use of C's arrow operator for structure member access 23⟩ ≡

```
    while ($look_for_identifiers =~ m/^(.*?)\-\>([A-Za-z_][A-Za-z0-9_]*)(.*?)$/) {
        $look_for_identifiers = $1.$3;
        note_usage_of_structure_member($2, $i);
    }
```

This code is used in §20.

§24. Since we know all the function names by this point...

define `$debug_identifier_detection 0`

⟨Is this identifier a function whose name we recognise? 24⟩ ≡

```
    if (exists $functions_line{$identifier}) {
        if ($debug_identifier_detection == 1) { print STDERR "ID: ", $identifier, "\n"; }
        if ($identifier eq "main") { next; }                    which of course isn't called in the web
        my $section_defining_function =
            $section_leafname[$line_sec[$functions_line{$identifier}]];
        $functions_usage_count{$identifier}++;
        $function_usage_hash{$identifier.'+'.$section_chap[$line_sec[$i]]} ++;
        if ($current_section_leafname ne $section_defining_function) {
            $functions_usage_section_list{$identifier} .=
                ":".$section_chap[$line_sec[$i]]."-".$current_section_leafname;
            $functions_usage_concisely_described{$identifier} .=
                ":".$current_section_number;
            $functions_usage_verbosely_described{$identifier} .=
                language_comment("called by Chapter ".
                    $section_chap[$line_sec[$i]]."/".$current_section_leafname.
                    " line ".$line_source_file_line[$i]);
        }
        if ($debug_identifier_detection == 1) {
            print STDERR "Added fn to code area $section_chap[$line_sec[$i]]\n";
        }
    }
```

This code is used in §20.

§**25.** For more on what these macros do, see the Inform source. `CREATE(T)` is a macro creating a structure of type `T`, which is defined with:

```
@d CREATE(type_name) ...
```

This definition shouldn't count as a creation of anything, but if `T` is anything else then it does. The sole point of this check is to warn us if we've accidentally created instances of the same structure in more than one section of Inform's web.

Macros such as `ALLOW_ARRAY(scene_data)` are used in Inform to set up what is legal when reading I6 template code: for instance, this would enable

```
{-array:scene_data}
```

the effect being that Inform's `compile_scene_data_array` function would be called whenever this token in an I6 template was expanded. (We have to take note of that here since the use of a macro disguises that a function call is being made.)

⟨Detect some of NI's fruitier macros 25⟩ ≡

```
    if ($look_for_identifiers =~ m/CREATE\((.*?)\)/) {
        if ($1 ne "type_name") {
            if ((exists ($structure_CREATE_in_section{$1})) &&
                ($structure_CREATE_in_section{$1} ne $section_leafname[$line_sec[$i]])) {
                add_error_to_section($current_section_number,
                    "Use of CREATE($1) in multiple sections");
            }
            $structure_CREATE_in_section{$1} = $section_leafname[$line_sec[$i]];
        }
    }
    while ($look_for_identifiers =~ m/^(.*?)ALLOW_([A-Z]+)\(([A-Za-z_0-9:_]*)\)(.*)$/) {
        $identifier = $3;
        $metalanguage_type = $2;
        $look_for_identifiers = $1.$4;
        $identifier =~ s/::/__/g;
        if (($metalanguage_type eq "CALL") ||
            ($metalanguage_type eq "CALLV")) {
            $dot_i6_identifiers{$identifier} = 1;
        }
        if ($metalanguage_type eq "ARRAY") {
            $dot_i6_identifiers{$identifier."_array"} = 1;
        }
        if ($metalanguage_type eq "ROUTINE") {
            $dot_i6_identifiers{$identifier."_routine"} = 1;
        }
    }
```

This code is used in §20.

§**26.** Keep track of each distinct section in which a given structure member is used:

```
sub note_usage_of_structure_member {
    my $member_name = $_[0];
    my $line_number = $_[1];
    my $found_in = "-".$section_leafname[$line_sec[$line_number]].":";
    if ($member_usage_section_list{$member_name} =~ m/$found_in/) { return; }
    $member_usage_section_list{$member_name} .= $found_in;
}
```

§27. Determine which sections call which other sections. We now know all of the functions, and where all of the function calls occur. This means we can determine (i) the actual scope of a function and (ii) how the "used by" and "uses" parts of a section's `@Interface` ought to read, if it were going to have them.

```
sub find_actual_section_usage {
    my $sn;
    for ($sn=0; $sn<$no_sections; $sn++) {
        my %usages = ();
        ⟨Check that the section's declared I6 template interface is correct 32⟩;
        foreach $f (sort keys %functions_line) {
            if ($section_leafname[$line_sec[$functions_line{$f}]] eq $section_leafname[$sn]) {
                ⟨Check that this function is properly declared, and add sections calling it to the usages hash 28⟩;
            }
        }
        if ($section_declared_a_service[$sn] == 1) { %usages = (); }
        ⟨Translate the usages hash into an unsorted form of the correct used-by block 29⟩;
    }
    for ($sn=0; $sn<$no_sections; $sn++)
        ⟨Deduce the correct uses block from the collection of all correct used-by blocks 30⟩;
    for ($sn=0; $sn<$no_sections; $sn++)
        ⟨Tidy up and finalise the correct uses and used-by blocks 31⟩;
}
```

§28. A neat trick here is that we knock the function's own chapter out of its usage list with a search and replace, and then look for chapter markers in what remains: if there are any, then the usage list must have contained sections from other chapters.

⟨Check that this function is properly declared, and add sections calling it to the usages hash 28⟩ ≡

```
    my $owning_chapter = $section_chap[$line_sec[$functions_line{$f}]];
    my $cp = $functions_usage_section_list{$f};
    $cp =~ s/\:$owning_chapter\-//g;
    my $actual_scope = "";
    if ($functions_usage_section_list{$f} ne "") { $actual_scope = "**"; }
    if ($f eq "main") { $actual_scope = "*****"; }
    elsif (exists $dot_i6_identifiers{$f}) {
        $actual_scope = "****";
        check_section_declares_template_I6($sn, $f);
    } elsif ($cp =~ m/\:\d+\-/) { $actual_scope = "***"; }
    check_function_declared_correct_scope($sn, $f, $actual_scope);
    $cp = $functions_usage_section_list{$f};
    while ($cp =~ m/^(\:\d+\-[^:]*)(.*)$/) {
        $usages{$1}++;
        $cp = $2;
    }
```

This code is used in §27.

§29. The usages hash is really just used as a set of keys here, and we concatenate it into a single string. (The list is unsorted but at least can't contain duplicates.) We tack a colon onto the end for convenience of the next search, but after that we'll remove it again.

⟨Translate the usages hash into an unsorted form of the correct used-by block 29⟩ ≡

```
$section_correct_used_by_block[$sn] = "";
foreach $u (sort keys %usages) {
    $section_correct_used_by_block[$sn] .= $u;
}
$section_correct_used_by_block[$sn] .= ":";
```

This code is used in §27.

§30. The uses blocks form a sort of transpose of the used-by blocks, if we think of these data structures as matrices rather than arrays: and as with all matrix-esque calculations, we end up with quadratic running time in the dimensions of the matrix. That is, if there are s sections, then the following takes $O(s^2)$ time, but since in practice $s < 200$ this is quite bearable.

⟨Deduce the correct uses block from the collection of all correct used-by blocks 30⟩ ≡

```
$section_correct_uses_block[$sn] = "";
my $j;
for ($j=0; $j<$no_sections; $j++) {
    if ($section_correct_used_by_block[$j] =~ m/\-$section_leafname[$sn]\:/) {
        if ($section_declared_a_service[$j] != 1) {
            $section_correct_uses_block[$sn]
                .= ":".$section_chap[$j]."-".$section_leafname[$j];
        }
    }
}
```

This code is used in §27.

§31. Note the promised removal of the terminating colons in the used-by blocks:

⟨Tidy up and finalise the correct uses and used-by blocks 31⟩ ≡

```
$section_correct_used_by_block[$sn] =~ s/\:$//;
$section_correct_used_by_block[$sn] =
    section_list_sort($section_correct_used_by_block[$sn]);
$section_correct_uses_block[$sn] =
    section_list_sort($section_correct_uses_block[$sn]);
```

This code is used in §27.

§32. If a section in a C-for-Inform web which uses an `@Interface` is going to name any I6 template things, well, they'd better be right: even in Weak mode we're not going to forgive any error in this.

⟨Check that the section's declared I6 template interface is correct 32⟩ ≡

```
my $a = $section_I6_template_identifiers[$sn];
while ($a =~ m/^(.*?)\:(.*)$/) {
    $a = $2;
    if (not(exists $dot_i6_identifiers{$1})) {
        add_error_to_section($sn, "Bad scope: $1 isn't a {-} command as claimed");
    }
    if ($section_leafname[$line_sec[$functions_line{$1}]] ne $section_leafname[$sn]) {
        add_error_to_section($sn, "Bad scope: $1 isn't defined in this section");
    }
}
```

This code is used in §27.

§**33. Structures.** We now come to the second part of the checking code for C-like sections: the part which verifies that structures and their members are used "correctly". We don't come to the party with nothing in hand, though: we have already found all the `typedef` structure declarations, so we know what the names of the structures are and on what line they are created; and we have kept an eye out, if in C-for-Inform mode, for any use of the `CREATE(S)` memory management macro as applied to a structure `S`.

§**34.** First: which section rightfully "owns" a structure? First, anyone with the power to `CREATE` has dominion; if nobody does, then anyone who claims the right in his `@Interface` is allowed to have it; and if nobody even claims the structure, well then, it belongs to whichever section declared its `typedef`.

```
sub find_structure_ownership {
    my $struc;
    foreach $struc (keys %structures) {
        if (exists $structure_CREATE_in_section{$struc}) {
            $structure_owner{$struc} = $structure_CREATE_in_section{$struc};
        } elsif (exists $structure_declared_owner{$struc}) {
            $structure_owner{$struc} =
                $structure_declared_owner{$struc};
        } else {
            $structure_owner{$struc} =
                $section_leafname[$line_sec[$structure_declaration_line{$struc}]];
        }
    }
}
```

§**35.** Second: which sections use which members, and by extension, which structures? Are any members entirely unused and consequently redundant?

```
sub establish_structure_usage {
    my $member;
    ⟨Check that any -analyse-structure command line setting wasn't mistyped 36⟩;
    foreach $member (sort keys %member_structures) {
        if (exists $member_usage_section_list{$member}) {
            my $msul = $member_usage_section_list{$member};
            while ($msul =~ m/^\-(.*?)\:(.*)$/) {
                my $observed_in = $1; $msul = $2;
                my $owners = $member_structures{$member};
                while ($owners =~ m/^(.*?)\:(.*)$/) {
                    my $owner = $1; $owners = $2;
                    my $susl_chunk = "-".$observed_in.":";
                    ⟨In -analyse-structure mode, throw an error for each member used in a foreign section 37⟩;
                    if (not ($structure_usage_section_list{$owner} =~ m/$susl_chunk/)) {
                        $structure_usage_section_list{$owner} .= $susl_chunk;
                    }
                }
            }
        } else {
            member_declared_but_not_used($member);
        }
    }
    ⟨Compose the correct structure declaration to appear in the Interface of each section 38⟩;
}
```

§36. As crude as this tool is, it was surprisingly helpful during a consolidation period when the code of NI was being tidied up into a more encapsulated form.

⟨Check that any -analyse-structure command line setting wasn't mistyped 36⟩ ≡

```
if ($analyse_structure_setting ne "") {
    if (not(exists($structures{$analyse_structure_setting}))) {
        inweb_fatal_error("no such structure: ".$analyse_structure_setting);
    }
    print "The structure $analyse_structure_setting is owned by ",
        $structure_owner{$analyse_structure_setting}, "\n";
    print "Members of the structure used from other sections are as follows:\n";
}
```

This code is used in §35.

§37. Perhaps these aren't really errors, but it does make for a tidy end to the process after we're done.

⟨In -analyse-structure mode, throw an error for each member used in a foreign section 37⟩ ≡

```
if (($analyse_structure_setting ne "") &&
    ($analyse_structure_setting eq $owner)) {
    if ($observed_in ne $structure_owner{$owner}) {
        inweb_error($member.": Chapter ".
            $section_chap[$section_number_from_leafname{$observed_in}].
            "/".$observed_in);
    }
}
```

This code is used in §35.

§38. The following assumes that structures are not used in the Appendices, which of course is true since these do not contain code.

⟨Compose the correct structure declaration to appear in the Interface of each section 38⟩ ≡

```
my $sec_number;
for ($sec_number=0; $sec_number<$no_sections; $sec_number++) {
    my $declaration = "";
    my $struc;
    foreach $struc (sort keys %structures) {
        if ($structure_owner{$struc} eq $section_leafname[$sec_number]) {
            my $structure_billing = "";
            ⟨Work out the billing for this structure 39⟩;
            $declaration .= $structure_billing;
        }
    }
    $section_correct_structures_block[$sec_number] = $declaration;
}
```

This code is used in §35.

§39. What we declare for the structure is: whether private or public, that is, open for other sections to access its members; whether it is `typedef`d by a section other than its rightful owner (which is allowed but deprecated, in terms of `NI`); and if it is public, then which other sections have action.

⟨Work out the billing for this structure 39⟩ ≡

```
    my %usage_list = ();
    my %shorter_usage_list = ();
    my $structure_used_externally = 0;
    my $deviant_ownership_note = "";
    if ($section_leafname[$line_sec[$structure_declaration_line{$struc}]]
        ne $structure_owner{$struc}) {
        $deviant_ownership_note =
            ": typedef in $section_leafname[$line_sec[$structure_declaration_line{$struc}]]";
    }
    ⟨Fill the usage list hash with the names of external sections accessing structure 40⟩;
    if ($structure_used_externally == 0) {
        $structure_billing = "-- Owns struct $struc (private$deviant_ownership_note)\n";
        $private_structures++;
    } else {
        $structure_billing = "-- Owns struct $struc (public$deviant_ownership_note)\n";
        $shared_structures++;
        foreach $k (sort keys %usage_list) {
            $structure_billing .= $usage_list{$k};
            $structure_ownership_summary{$struc} .= $shorter_usage_list{$k} . " ";
        }
    }
```

This code is used in §38.

§40.

⟨Fill the usage list hash with the names of external sections accessing structure 40⟩ ≡

```
    my $susl = $structure_usage_section_list{$struc};
    while ($susl =~ m/^\-(.*?)\:(.*)$/) {
        my $client_section = $1; $susl = $2;
        if ($client_section ne $section_leafname[$sec_number]) {
            $structure_sharings++;
            $structure_used_externally = 1;
            $usage_list{sprintf("%05d", $section_number_from_leafname{$client_section})}
                .= "   !- shared with Chapter ".
                $section_chap[$section_number_from_leafname{$client_section}].
                "/".$client_section."\n";
            $shorter_usage_list{sprintf("%05d", $section_number_from_leafname{$client_section})}
                .= $section_sigil[$section_number_from_leafname{$client_section}];
        }
    }
```

This code is used in §39.

§**41. Hacking with section list strings.** We take a section list string to pieces, sort it and reassemble it in sorted order. Don't look so suspicious: it works.

```
sub section_list_sort {
    my $list = $_[0];
    my %lines_unpacked = ();
    my $this_line;
    my $account = "";
    while ($list =~ m/^\:(\d+)\-([^:]*)(.*)$/) {
        $this_line = $prefix. "Chapter ". $1. "/". $2. "\n";
        $lines_unpacked{sprintf("%04d", $section_number_from_leafname{$2})} = ":".$1."-".$2;
        $list = $3;
    }
    foreach $this_line (sort keys %lines_unpacked) {
        $account .= $lines_unpacked{$this_line};
    }
    return $account;
}
```

§**42.** To unpack a section list is to reconstruct it as a sequence of lines in sorted order and with a slightly different format.

```
sub section_list_unpack {
    my $prefix = $_[0];
    my $list = $_[1];
    my %lines_unpacked = ();
    my $this_line;
    my $account = "";
    while ($list =~ m/^\:(\d+)\-([^:]*)(.*)$/) {
        $this_line = $prefix. "Chapter ". $1. "/". $2. "\n";
        $lines_unpacked{sprintf("%04d", $section_number_from_leafname{$2})} = $this_line;
        $list = $3;
    }
    foreach $this_line (sort keys %lines_unpacked) {
        $account .= $lines_unpacked{$this_line};
    }
    return $account;
}
```

§43. Applying the Strict Usage Rules of a vengeful god.

```
sub check_interface_declarations_for_uses_and_used_by {
    my $sn;
    if ($bibliographic_data{"Strict Usage Rules"} eq "Off") { return; }
    if ($bibliographic_data{"Declare Section Usage"} eq "Off") { return; }
    for ($sn=0; $sn<$no_sections; $sn++) {
        if ($section_declared_used_by_block[$sn] ne
            $section_correct_used_by_block[$sn]) {
            $em = "Interface error:\n"
                .section_list_unpack(
                    "%    Says used by ", $section_declared_used_by_block[$sn])
                .section_list_unpack(
                    "-- Used by ", $section_correct_used_by_block[$sn]);
            add_error_to_section($sn, $em);
        }
        if ($section_declared_uses_block[$sn] ne
            $section_correct_uses_block[$sn]) {
            $em = "Interface error:\n"
                .section_list_unpack(
                    "%    Says uses ", $section_declared_uses_block[$sn])
                .section_list_unpack(
                    "-- Uses ", $section_correct_uses_block[$sn]);
            add_error_to_section($sn, $em);
        }
    }
}
```

§44. And similarly for the structures part of the `@Interface:`...

```
sub check_interface_declarations_for_structures {
    my $sn;
    for ($sn=0; $sn<$no_sections; $sn++) {
        if ($section_declared_structure_interface[$sn] ne
            $section_correct_structures_block[$sn]) {
            add_error_to_section($sn,
                "Incorrect structure billing: should be...\n".
                $section_correct_structures_block[$sn].
                "...and not...\n".
                $section_declared_structure_interface[$sn]);
        }
    }
}
```

§45. If we're being Strict, functions must also be marked with their correct scopes:

```
sub check_function_declared_correct_scope {
    my $sn = $_[0];
    my $f = $_[1];
    my $actual_scope = $_[2];
    if ($bibliographic_data{"Namespaces"} eq "On") {
        if ($f ne "main") {
            if (($actual_scope ne "") && ($functions_declared_scope{$f} eq "")) {
                add_error_to_section($sn,
                    "Begin externally called, function $f should belong to a :: namespace");
                return;
            }
            if (($actual_scope eq "") && ($functions_declared_scope{$f} ne "")) {
                add_error_to_section($sn,
                    "Begin internally called, function $f must not belong to a :: namespace");
                return;
            }
        }
        $functions_declared_scope{$f} = $actual_scope;
        return;
    }
    if ($bibliographic_data{"Strict Usage Rules"} eq "Off") { return; }
    if ($actual_scope ne $functions_declared_scope{$f}) {
        add_error_to_section($sn,
            "Bad scope: $f should be /".$actual_scope."/ not /".
                $functions_declared_scope{$f}."/");
    }
    if ($f =~ m/^([A-Z].*__)(.*?)$/) {
        my $declared = $1;
        my $within = $section_namespace[$sn];
        $within =~ s/::/__/g;
        if ($within eq "") {
            $fcc = restore_quadpoint($f);
            add_error_to_section($sn,
                "Bad scope: $fcc declared outside of any namespace");
        } elsif (not ($declared =~ m/^$within/)) {
            $fcc = restore_quadpoint($f); $wcc = restore_quadpoint($within);
            add_error_to_section($sn,
                "Bad scope: $fcc not allowed inside the section's namespace $wcc");
        }
    }
}

sub restore_quadpoint {
    my $f = $_[0];
    $f =~ s/__/::/g;
    return $f;
}
```

§**46.** The following is applied only in C-for-Inform mode; in ordinary C, no such functions occur.

```
sub check_section_declares_template_I6 {
    my $sn = $_[0];
    my $f = $_[1];
    if ($bibliographic_data{"Strict Usage Rules"} eq "Off") { return; }
    f =  s//::/g;
    if (not ($section_I6_template_identifiers[$sn] =~ m/$f/)) {
        add_error_to_section($sn,
            "Bad scope: $f isn't declared:\n-- Defines {-callv:".$f."}\n");
    }
}
```

§**47.** It is, of course, a Sin to declare structures with member names that are never actually used. Unless of course we are in a messy program which uses them in concealed ways because of macro expansion – hence the exemption clause.

```
sub member_declared_but_not_used {
    my $member = $_[0];
    if ($bibliographic_data{"Strict Usage Rules"} eq "Off") { return; }
    if (exists($members_allowed_to_be_unused{$member})) { return; }
    inweb_error("structure(s) '".$member_structures{$member}."' has or have member '".
        $member."' declared but not used");
}
```

§**48.** `inweb` prefers a tidy setup where the same member name is not used in multiple structures, and in strict mode, it insists of this: that is, where `p->drivel` identifies the type of `p` since `drivel` is a member in only one possible structure. The following routine polices that rule.

```
sub check_uniqueness_of_structure_members {
    my $member;
    if ($bibliographic_data{"Strict Usage Rules"} eq "Off") { return; }
    foreach $member (sort keys %member_structures) {
        my $owner_count = 0;
        my $x = $member_structures{$member};
        while ($x =~ m/^(.*?)\:(.*)$/) { $x = $2; $owner_count++; }
        if ($owner_count > 1) {
            inweb_error("element '".$member."' belongs to multiple structures: ".
                $member_structures{$member});
        }
    }
}
```

§**49. Interface errors.** When an interface error comes to light, we call:

```
sub add_error_to_section {
    my $section_number = $_[0];
    my $error_text = $_[1];
    if ($error_text eq "") { print "*** Bad error text ***\n"; exit(1); }
    $section_errors[$section_number] .= $error_text."\n";
}
```

§**50.** So that at the end of the checking process, we can report and take action:

```
sub report_any_interface_errors_found {
    my $infractions = 0;
    my $sn;
    for ($sn=0; $sn<$no_sections; $sn++) {
        if ($section_errors[$sn] ne "") {
            inweb_error("interface errors on $section_sigil[$sn] ($section_leafname[$sn])");
            print STDERR $section_errors[$sn];
            $infractions++;
        }
    }
    if ($infractions > 0) { inweb_fatal_error("halting because of section errors"); }
}
```