INWEB

The Program

Chapter 3

Build 4/090319 Graham Nelson

3 Outputs

`3/anal`: *The Analyser.w* Miscellaneous useful (or anyway, formerly useful) checks to carry out on the source code.

`3/swarm`: *The Swarm.w* To feed multiple output requests to the weaver, and to present weaver results, and update indexes or contents pages.

`3/weave`: *The Weaver.w* To weave a portion of the code into instructions for TeX.

`3/bnf`: *Backus-Naur Form.w* To weave any collated BNF grammar from the web into a nicely typeset form.

`3/tang`: *The Tangler.w* To write a portion of the code in a compilable form.

`3/plan`: *Programming Languages.w* To characterise the relevant differences in behaviour between the various programming languages we support: for instance, how comments are represented.

`3/cfori`: *C for Inform.w* To provide a convenient extension to C syntax for the C-for-Inform language, which is likely never to be used for any program other than the Inform 7 compiler.

The Analyser 3/anal

Purpose

Miscellaneous useful (or anyway, formerly useful) checks to carry out on the source code.

§**1. Dot files for dependency graphs.** The target `0` as always means the entire web, so that we have a dependency graph of each chapter; otherwise we must specify a particular chapter (not a section, not an appendix).

```
sub compile_graphs {
    my $sigil = $_[0];
    if ($sigil eq "0") {
        my $ch;
        for ($ch=0; $ch<$no_chapters; $ch++) { compile_graph($ch); }
    } elsif ($sigil =~ m/^\d+$/) {
        compile_graph(eval($sigil));
    } else {
        inweb_fatal_error("can't compile dependency graph(s) for target $sigil");
    }
    print "Dot files written\n";
}
```

§**2.** Thus the following compiles the graph for a single chapter, converting it on request into a PNG image.

```
sub compile_graph {
    my $ch = $_[0];
    my $dotfile = compile_chapter_graph($ch);
    my $pngfile = pathname_to_png_of_dot_file($ch);
    if ($convert_graphs_switch == 1) {
        system($dot_utility_path." -Tpng \"".$dotfile."\" -o \"".$pngfile."\"");
    }
}
```

§**3.** The dot files are the sources of the graph illustrations: they need to be compiled by the `dot` utility to turn them into actual images. We keep the dot files in the `Tangled` folder, as they are essentially generated from code, but the images they turn into are in the `Figures` folder, since the expectation is that they'll be used as illustrations in the woven form of the web.

```
sub pathname_to_dot_file {
    my $cn = $_[0];
    return $web_setting."Tangled/Chapter-".$cn."-Dependencies.dot";
}
sub pathname_to_png_of_dot_file {
    my $cn = $_[0];
    return $web_setting."Figures/Chapter-".$cn."-Dependencies.png";
}
```

§**4.** This is all pretty straightforward: see the `dot` user manual for details of the format.

```
sub compile_chapter_graph {
    my $cn = $_[0];
    my $i;
    $dotname = pathname_to_dot_file($cn);
    open DOTFILE, ">".$dotname or die "Can't open dot file $dotname for output";
    ⟨Start a directed graph 5⟩;
    ⟨Declare the vertices 7⟩;
    ⟨Declare the edges 9⟩;
    ⟨End the directed graph 6⟩;
    close DOTFILE;
    return $dotname;
}
```

§**5.** The start is always the same:

⟨Start a directed graph 5⟩ ≡

```
    print DOTFILE "digraph L0 {\n";
    print DOTFILE "    size = \"8,8\";\n";
    print DOTFILE "    ordering = out;\n";
    print DOTFILE "    compound = true;\n";
    print DOTFILE "    node [shape = box];\n";
```

This code is used in §4.

§**6.** And so is the end:

⟨End the directed graph 6⟩ ≡

```
    print DOTFILE "}\n";
```

This code is used in §4.

§**7.** In between, we start with vertex declarations: one for each section in the chapter, plus one for the template interpreter.

⟨Declare the vertices 7⟩ ≡

```
    my $i;
    $current_class = -1; $i6_controller = -1;
    for ($i=0; $i<$no_sections; $i++) {
        if ($section_chap[$i] != $cn) { next; }
        ⟨Declare a vertex for this section 8⟩;
    }
    ⟨Make an orchid-coloured circle to represent the template interpreter 12⟩;
    if ($current_class >= 0) { print DOTFILE "}\n"; }
```

This code is used in §4.

§**8.** We draw a subgraph box around all vertices in a given equivalence class, which is neat if we're drawing a big multi-chapter graph, but in fact we aren't doing that so instead `$equivalence_class[$i]` is always 0 for now, and the effect is largely decorative – though note that the special vertex for the I6T interpreter, in the case of graphing Inform, lies outside the box (and rightly, since it makes function calls from out of the sky).

⟨Declare a vertex for this section 8⟩ ≡

```
$caption = $section_sigil[$i];
if ($caption eq "14/meta") { $i6_controller = $i; }
if ($section_sigil[$i] ne "") { $caption = $section_sigil[$i]; }
if ($current_class != $equivalence_class[$i]) {
    if ($current_class >= 0) { print DOTFILE "}\n"; }
    $current_class = $equivalence_class[$i];
    print DOTFILE "subgraph cluster", $cn, " {\n";
    print DOTFILE "label=\"Chapter ", $cn, "\";\n";
}
print DOTFILE "    n", $i, " [label=\"", $caption, "\"];\n";
if (($section_I6_template_identifiers[$line_sec[$i]] ne "") ||
    ($i6_controller == $i)) {
    $section_I6_template_identifiers[$line_sec[$i]] = "";
    if ($i6_controller != $i) { $dot_i6_calls{$i} = "i6"; }
}
```

This code is used in §7.

§**9.**

⟨Declare the edges 9⟩ ≡

```
my $i;
for ($i=0; $i<$no_sections; $i++) {
    if ($section_chap[$i] != $cn) { next; }
    ⟨Declare the directed edges outbound from this vertex 10⟩;
}
```

This code is used in §4.

§**10.**

⟨Declare the directed edges outbound from this vertex 10⟩ ≡

```
my $x = $section_correct_uses_block[$i];
my $no_outbound_calls = 0;
while ($x =~ m/^\:(.*?)\-(.*?\.w)(.*)$/) {
    $x = $3;
    $to_section = $section_number_from_leafname{$2};
    $from_chapter = $section_chap[$i];
    $to_chapter = $section_chap[$to_section];
    if ($from_chapter == $to_chapter) {
        $uses_this[$no_outbound_calls] = $to_section;
        $uses_this_section[$no_outbound_calls] = 1;
        $no_outbound_calls++;
    } else {
        $uses_this[$no_outbound_calls] = $to_chapter;
        $uses_this_section[$no_outbound_calls] = 0;
        $no_outbound_calls++;
    }
}
```

⟨Make black arrows for all the calls out of this section into others on the graph 11⟩;
⟨Make an arrow representing a template interpreter invocation if necessary 13⟩;

This code is used in §9.

§**11.** This makes lines like `n3 -> {n2, n4, n5}`, meaning arrows run from node 3 to each of nodes 2, 4 and 5.

⟨Make black arrows for all the calls out of this section into others on the graph 11⟩ ≡

```
if ($no_outbound_calls > 0) {
    print DOTFILE "    n", $i, " -> ";
}
if ($no_outbound_calls > 1) {
    print DOTFILE "{ ";
}
for ($z=0; $z<$no_outbound_calls; $z++) {
    if ($uses_this_section[$z] == 1) {
        print DOTFILE "n", $uses_this[$z], " ";
    }
}
if ($no_outbound_calls > 1) {
    print DOTFILE "} ";
}
if ($no_outbound_calls > 0) {
    print DOTFILE "\n";
}
```

This code is used in §10.

§**12.** For use with `NI` only:

⟨Make an orchid-coloured circle to represent the template interpreter 12⟩ ≡

```
print DOTFILE "    i6 [shape=circle] [color=orchid] [label=\".i6\"];\n";
```

This code is used in §7.

§**13.** Orchids come in lots of colours, but this one is quite a pleasing pastel purple, anyway.

⟨Make an arrow representing a template interpreter invocation if necessary 13⟩ ≡

```
if ($dot_i6_calls{$i} ne "") {
    print DOTFILE "    ", $dot_i6_calls{$i}, " -> n", $i, " [color=orchid];\n";
}
```

This code is used in §10.

§**14.** And so that this does, somehow, form part of the call graph itself:

⟨On the section owning the template interpreter, make an arrow pointing to it 14⟩ ≡

```
my $i;
for ($i=0; $i<$no_sections; $i++) {
    if ($section_chap[$i] != $cn) { next; }
    if ($section_sigil[$i] eq "14/meta") {
        print DOTFILE "    n", $i6_controller, " -> i6 [color=orchid];\n";
    }
}
```

This code is used in §.

§15. The section catalogue. Quite a useful snapshot of the sections, and also of the data structures used in a big C-like project.

```
sub catalogue_the_sections {
    my $sigil = $_[0];
    my $functions_too = $_[1];
    my $only = -1;
    my $cn = -1;
    if ($sigil eq "0") { $only = -1; }
    elsif ($sigil =~ m/^\d+$/) { $only = eval($sigil); }
    else { inweb_fatal_error("can't catalogue target $sigil"); }
    for ($i=0; $i<$no_sections; $i++) {
        if (($only != -1) && ($only != $section_chap[$i])) { next; }
        ⟨Produce dividing bars between chapters 16⟩;
        ⟨Produce catalogue line for this section 17⟩;
        if ($functions_too == 1) ⟨Produce list of functions owned by this section 19⟩
        else ⟨Produce list of data structures owned by this section 18⟩;
    }
}
```

§16.

⟨Produce dividing bars between chapters 16⟩ ≡

```
    if ($cn != $section_chap[$i]) {
        if ($cn >= 0) { print sprintf("      %-9s  %-50s  \n", "--------", "--------"); }
        $cn = $section_chap[$i];
    }
```

This code is used in §15.

§17.

⟨Produce catalogue line for this section 17⟩ ≡

```
    if ($cn != 0) { $main_title = "Chapter ".$cn."/"; }
    else { $main_title = ""; }
    $main_title .= $section_leafname[$i];
    print sprintf("%4d  %-9s  %-50s  ",
        $section_extent[$i], $section_sigil[$i], $main_title);
    if ($functions_too == 1) {
        print $section_namespace[$i];
    } else {
        print $section_namespace[$i], " ";
        if ($functions_usage_count{"compare_word"} > 0) {
            print sprintf("CW:%3d   ", $functions_usage_count{"compare_word"});
        }
        foreach $struc (sort keys %structures) {
            if ($structure_owner{$struc} eq $section_leafname[$i]) {
                print $struc, "  ";
            }
        }
    }
    print "\n";
```

This code is used in §15.

§**18.** This does nothing for non-C-like languages, since the hash is empty.

⟨Produce list of data structures owned by this section 18⟩ ≡
```
    foreach $struc (sort keys %structures) {
        if ($structure_owner{$struc} eq $section_leafname[$i]) {
            if ($structure_ownership_summary{$struc} ne "") {
                print sprintf("      %-9s  %-50s  ", "", "");
                print $struc, ": ", $structure_ownership_summary{$struc}, "\n";
            }
        }
    }
```

This code is used in §15.

§**19.** And here we have the function catalogue:

⟨Produce list of functions owned by this section 19⟩ ≡
```
    foreach $f (sort keys %functions_line) {
        if ($f =~ m/__/) {
            if ($section_leafname[$line_sec[$functions_line{$f}]] eq $section_leafname[$i]) {
                $f =~ s/__/::/g;
                print sprintf("      %-9s  %-50s -> \n", "", $f);
            }
        }
    }
```

This code is used in §15.

§**20.** The void of eternity. Or what have you. Presuming that `void *` pointers are the road to a dusty death, we catalogue them here. (This is a leftover from the days when the main Inform source used `void *` pointers now and then, and the code was added here to help purge them from Inform by reporting them.)

```
sub catalogue_void_pointers {
    my $sigil = $_[0];
    if ($sigil ne "0") { inweb_fatal_error("can't catalogue voids for target $sigil"); }
    my $el;
    foreach $el (sort keys %member_types) {
        if ($member_types{$el} =~ m/ void\*/) {
            print $el, " in ", $member_structures{$el}, ": ", $member_types{$el}, "\n";
        }
    }
}
```

The Swarm 3/swarm

Purpose

To feed multiple output requests to the weaver, and to present weaver results, and update indexes or contents pages.

§**1. Swarming.** Let us hope this is a glittering cloud, like the swarm of locusts in the title of Chapter 25 of Laura Ingalls Wilder's *On the Banks of Plum Creek*: at any rate it does make quite a deal of individual PDFs.

Each individual weave returns an ID number, its "weave number", which can be used to look up the individual results. This is stored in the arrays:

(C5) `$chapter_weave_number[]` is either the ID of the completed weave of this individual chapter, or -1 if no weave was made from it.

(S18) `$section_weave_number[]` is either the ID of the completed weave of this individual section, or -1 if no weave was made from it.

```
sub weave_swarm {
    my $i;
    for ($i=0; $i < $no_sections; $i++) { $section_weave_number[$j] = -1; }
    for ($i=0; $i < $no_chapters; $i++) { $chapter_weave_number[$j] = -1; }
    if ($swarm_mode >= $SWARM_SECTIONS) {
        for ($i=0; $i < $no_sections; $i++) {
            if (($only_setting eq "") || ($section_sigil[$i] =~ m/^$only_setting\//)) {
                $section_weave_number[$i] = weave_sigil($section_sigil[$i], 0);
            }
        }
    }
    if (($web_is_chaptered == 1) && ($swarm_mode >= $SWARM_CHAPTERS)) {
        for ($i=0; $i < $no_chapters; $i++) {
            if ($only_setting ne "") {
                if ($chapter_sigil[$i] ne $only_setting) { next; }
            }
            $chapter_weave_number[$i] = weave_sigil($chapter_sigil[$i], 0);
            if ($only_setting ne "") {
                $complete_web_weave_number = $chapter_weave_number[$i];
                $complete_PDF_leafname = $chapter_woven_pdf_leafname[$i];
            }
        }
    }
    if (($swarm_mode >= $SWARM_CHAPTERS) && ($only_setting eq "")) {
        $complete_web_weave_number = weave_sigil("0", 0);
    }
    weave_index_templates();
}
```

§**2.** After every swarm, we rebuild all the indexes specified in the Index Template list for the web:

```
sub weave_index_templates {
    ⟨Weave one or more index files 3⟩;
    ⟨Copy in one or more additional files to accompany the index 4⟩;
}
```

§**3.**

⟨Weave one or more index files 3⟩ ≡

```
    my $temp_list;
    my $leaf = "";
    if (exists ($bibliographic_data{"Index Template"})) {
        $temp_list = $bibliographic_data{"Index Template"};
    } else {
        if ($web_is_chaptered == 1) {
            $temp_list = $path_to_inweb_setting.'inweb/Materials/chaptered-index.html';
        } else {
            $temp_list = $path_to_inweb_setting.'inweb/Materials/unchaptered-index.html';
        }
        $leaf = "index.html";
    }
    while ($temp_list ne "") {
        my $this_temp;
        if ($temp_list =~ m/^(.*?)\s*\,\s*(.*)$/) {
            $temp_list = $2; $this_temp = $1;
        } else {
            $this_temp = $temp_list; $temp_list = "";
        }
        if ($leaf eq "") {
            $leaf = $this_temp;
            if ($leaf =~ m/^.*\/(.*?)$/) { $leaf = $1; }
        }
        print "Weaving index file: Woven/$leaf\n";
        weave_contents_from_template($this_temp, $leaf);
        $leaf = "";
    }
```

This code is used in §2.

§**4.** The idea here is that an HTML index may need some binary image files to go with it, for instance.

⟨Copy in one or more additional files to accompany the index 4⟩ ≡

```
my $copy_list = $path_to_inweb_setting.'inweb/Materials/download.gif'.
    ', '.$path_to_inweb_setting.'inweb/Materials/lemons.jpg';
if (exists ($bibliographic_data{"Index Extras"})) {
    $temp_list = $bibliographic_data{"Index Extras"};
}
my $path_to_copy_binaries = $web_setting."Woven";
while ($copy_list ne "") {
    my $this_file;
    if ($copy_list =~ m/^(.*?)\s*\,\s*(.*)$/) {
        $copy_list = $2; $this_file = $1;
    } else {
        $this_file = $copy_list; $copy_list = "";
    }
    my $leaf = $this_file;
    if ($leaf =~ m/^.*\/(.*?)$/) { $leaf = $1; }
    print "Copying additional index file: Woven/$leaf\n";
    system("cp '".$this_file."' '".$path_to_copy_binaries."'");
}
```

This code is used in §2.

§**5.** The following is where an individual weave task begins, whether it comes from the swarm, or has been specified at the command line (in which case the call comes from Program Control).

```
sub weave_sigil {
    my $request = $_[0];                                  The sigil of the target to be weaved
    my $open_afterwards = $_[1];                          Open the PDF in the host OS's viewer
    my $cover_sheet_flag = 0;                             Shall we have one?
    my $weave_section_match = '';                         Reg exp for sigil to match
    my $tex_file_leafname;                                What to call the resulting TeX file
    my $path_to_loom = $web_setting."Woven/";
    ⟨Translate the request sigil into details of what to weave 6⟩;
    weave_source($path_to_loom.$tex_file_leafname, $cover_sheet_flag, $weave_section_match);
    if ($lines_woven == 0) {
        inweb_fatal_error("empty weave request: $request");
    }
    my $wtn = run_woven_source_through_tex($path_to_loom.$tex_file_leafname,
        $open_afterwards, $cover_sheet_flag, $weave_section_match);
    ⟨Report on the outcome of the weave to the console 7⟩;
    return $wtn;
}
```

§**6.** From the sigil, we determine where to put the resulting TEX file, whether to make a PDF from it (always, at present), whether to open that PDF in the operating system (depends on the -open-pdf switch being used), whether to include a cover sheet (yes unless it's for just a single section), and which lines of the web should contribute to the source. We represent this last by supplying a regular expression which the sigil of the section owning the line should match: for instance, requiring this to match `11/` catches all of the lines in Chapter 11.

⟨Translate the request sigil into details of what to weave 6⟩ ≡

```
if ($request eq "0") {
    $weave_section_match = '.*';
    $booklet_title = "Complete Program";
    $tex_file_leafname = "Complete.tex";
    $cover_sheet_flag = 1;
} elsif ($request =~ m/^\d+$/) {
    my $cn = eval($request);
    $weave_section_match = '^'.$cn.'\/';
    $booklet_title = "Chapter ".$cn;
    $tex_file_leafname = "Chapter-".$cn.".tex";
    $cover_sheet_flag = 1;
} elsif ($request =~ m/^[A-O]$/) {
    my $cn = eval($request);
    $weave_section_match = '^'.$cn.'\/';
    $booklet_title = "Appendix ".$request;
    $tex_file_leafname = "Appendix-".$request.".tex";
    $cover_sheet_flag = 1;
} elsif ($request =~ m/^P$/) {
    my $cn = eval($request);
    $weave_section_match = '^'.$cn.'\/';
    $booklet_title = "Preliminaries";
    $tex_file_leafname = "Preliminaries.tex";
    $cover_sheet_flag = 1;
} elsif ($request =~ m/^(\S+?)\/(\S+)$/) {
    my $cn = eval($request);
    $weave_section_match = '^'.$1.'\/'.$2.'$';
    $booklet_title = $request;
    my $srequest = $request;
    $srequest =~ s/\//-/;
    $tex_file_leafname = $srequest.".tex";
    $cover_sheet_flag = 0;
} else {
    inweb_fatal_error("unknown weave request: $request");
}
```

This code is used in §5.

§**7.** Each weave results in a compressed one-line printed report:

⟨Report on the outcome of the weave to the console 7⟩ ≡

```
print "[", $request, ": ", $page_count[$wtn], "pp ",
    $pdf_size[$wtn]/1024, "K";
if ($overfull_hbox_count[$wtn] > 0) {
    print ", ", $overfull_hbox_count[$wtn], " overfull hbox(es)";
}
if ($tex_error_count[$wtn] > 0) {
    print ", ", $tex_error_count[$wtn], " error(s)";
}
print "]\n";
```

This code is used in §5.

§**8. Running TeX.** Although we are running `pdftex`, a modern variant of TEX, rather than the original, they are very similar as command-line tools; the difference is that the output is a PDF file rather than a DVI file, Knuth's original stab at the same basic idea.

In particular, we call it in "scrollmode" so that any errors whizz by rather than interrupting or halting the session. Because of that, we spool the output onto a console file which we can then read in and parse to find the number of errors actually generated. Prime among errors is the "overfull hbox error", a defect of TEX resulting from its inability to adjust letter spacing, so that it requires us to adjust the copy to fit the margins of the page properly. (In practice we get this here by having code lines which are too wide to display.)

```
sub run_woven_source_through_tex {
    my $tex_filename = $_[0];
    my $open_PDF = $_[1];
    my $with_cover_sheet = $_[2];
    my $weave_section_match = $_[3];

    my $path_to_tex = "";
    my $tex_leafname = "";
    my $console_filename = "";
    my $console_leafname = "";
    my $log_filename = "";
    my $pdf_filename = "";
    ⟨Work out these filenames and leafnames 9⟩;

    my $serious_error_count = 0;

    ⟨Call TeX and transcribe its output into a console file 10⟩;
    ⟨Read back the console file and parse it for error messages 11⟩;
    ⟨Remove the now redundant TeX, console and log files, to reduce clutter 12⟩;

    if ($open_PDF == 1) ⟨Try to open the PDF file in the host operating system 13⟩;
    return $no_weave_targets++;                    Return the weave ID number for this run of TEX
}
```

§**9.**

⟨Work out these filenames and leafnames 9⟩ ≡

```
$tex_leafname = $tex_filename;
if ($tex_leafname =~ m/^(.*)\/(.*?)$/) { $path_to_tex = $1; $tex_leafname = $2; }
$console_leafname = $tex_leafname; $console_filename = $tex_filename;
$console_leafname =~ s/\.tex$/.console/; $console_filename =~ s/\.tex$/.console/;
$log_filename = $tex_filename; $log_filename =~ s/\.tex$/.log/;
$pdf_filename = $tex_filename; $pdf_filename =~ s/\.tex$/.pdf/;
```

This code is used in §8.

§**10.**

⟨Call TeX and transcribe its output into a console file 10⟩ ≡

```
my $set_cwd = "";
if ($path_to_tex ne "") { $set_cwd = "cd \"".$path_to_tex."\"; "; }
$command = $set_cwd.$pdftex_configuration." -interaction=scrollmode \"".$tex_leafname."\" "
    .">\"".$console_leafname."\"";
system($command);
```

This code is used in §8.

§**11.** TEX helpfully reports the size and page count of what it produces, and we're not too proud to scrape that information out of the console file, besides the error messages (which begin with an exclamation mark in column 1).

⟨Read back the console file and parse it for error messages 11⟩ ≡

```
$overfull_hbox_count[$no_weave_targets] = 0;
$tex_error_count[$no_weave_targets] = 0;
$page_count[$no_weave_targets] = 0;
$pdf_size[$no_weave_targets] = 0;
open(CONSOLE, $console_filename) or die "no console file?";
while ($csl = <CONSOLE>) {
    if ($csl =~ m/Output written .*? \((\d+) page.*?(\d+) byte/) {
        $page_count[$no_weave_targets] = eval($1); $pdf_size[$no_weave_targets] = eval($2);
    }
    if ($csl =~ m/verfull \\hbox/) {
        $overfull_hbox_count[$no_weave_targets]++;
    } else {
        if ($csl =~ m/^\!/) {
            $tex_error_count[$no_weave_targets]++;
            $serious_error_count++;
        }
    }
}
close (CONSOLE);
```

This code is used in §8.

§**12.** The log file we never wanted, but TEX produced it anyway; it's really a verbose form of its console output. Now it can go. So can the console file and even the TEX source, since that was mechanically generated from the web, and so is of no lasting value. The one exception is that we keep the console file in the event of serious errors, since otherwise it's impossible for the user to find out what those errors were.

⟨Remove the now redundant TeX, console and log files, to reduce clutter 12⟩ ≡

```
if ($serious_error_count == 0) { system("rm \"".$console_filename."\""); }
system("rm \"".$log_filename."\"");
system("rm \"".$tex_filename."\"");
```

This code is used in §8.

§**13.** We often want to see the PDF immediately, so:

⟨Try to open the PDF file in the host operating system 13⟩ ≡

```
if ($open_command_configuration eq "") {
    inweb_error("no way to open PDF (see configuration file)", $pdf_filename);
} else {
    system($open_command_configuration." \"".$pdf_filename."\"");
}
```

This code is used in §8.

§**14. The Contents Interpreter.** This is a little meta-language all of its very own, with a stack for holding nested repeat loops, and a program counter and – well, and nothing else to speak of, in fact, except for the slightly unusual way that loop variables provide context by changing the subject of what is discussed rather than by being accessed directly.

define $TRACE_CI_EXECUTION 0 *For debugging*

```
sub weave_contents_from_template {
    my $path_to_template = $_[0];
    my $contents_page_leafname = $_[1];
    my $no_tlines = 0;
    ⟨Read in the source file containing the contents page template 15⟩;
    ⟨Open the contents page file to be constructed 16⟩;
    my $lpos = 0;                                   This is our program counter: a line number in the template
    $stack_pointer = 0;                             And this is our stack pointer for tracking of loops
    CYCLE: while ($lpos < $no_tlines) {
        my $tl = $tlines[$lpos++];                  Fetch the line at the program counter and advance
        $tl =~ m/^(.*?)\s*$/; $tl = $1;             Strip trailing spaces
        if ($TRACE_CI_EXECUTION == 1)
            ⟨Print line and contents of repeat stack 17⟩;
        if ($tl =~ m/^\s*\[\[(.*?)\]\]\s*$/) {
            my $command = $1;
            ⟨Deal with a Select command 18⟩;
            ⟨Deal with a Repeat command 19⟩;
            ⟨Deal with a Repeat End command 20⟩;
            print "Still here, ", $command, "\n";
        }
        ⟨Skip line if inside an empty loop 21⟩;
        ⟨Make substitutions of square-bracketed variables in line 24⟩;
        print CONTS $tl, "\n";                      Copy the now finished line to the output
    }
    close (CONTS);
}
```

§15. File handling.

⟨Read in the source file containing the contents page template 15⟩ ≡

```
if (not(open(TEMPL, $path_to_template))) {
    print "inweb: warning: unable to generate index because can't find template at ",
        $path_to_template, "\n";
    return;
}
while ($tl = <TEMPL>) {
    $tlines[$no_tlines++] = $tl;
}
close (TEMPL);
if ($TRACE_CI_EXECUTION == 1) {
    print "Read template <", $path_to_template, ">: ", $no_tlines, " line(s)\n";
}
```

This code is used in §14.

§16.

⟨Open the contents page file to be constructed 16⟩ ≡

```
my $path_to_contents = $web_setting."Woven/".$contents_page_leafname;
if (not(open(CONTS, '>'.$path_to_contents))) {
    print "inweb: warning: unable to generate index because can't open to write ",
        $path_to_contents, "\n";
    return;
}
```

This code is used in §14.

§17. The repeat stack and loops.

⟨Print line and contents of repeat stack 17⟩ ≡

```
my $j;
print sprintf("%04d: %tl\nStack:", $lpos-1, $tl);
for ($j=0; $j<$stack_pointer; $j++) {
    print " ", $j, ": ", $repeat_stack_variable[$j], "/", $repeat_stack_threshold[$j], " ",
        $repeat_stack_level[$j];
}
print "\n";
```

This code is used in §14.

§**18.** We start the direct commands with Select, which is implemented as a one-iteration loop in which the loop variable has the given section or chapter as its value during the sole iteration.

⟨Deal with a Select command 18⟩ ≡

```
if ($command =~ m/^Select (.*)$/) {
    my $sigil = $1;
    my $j;
    for ($j = 0; $j<$no_sections; $j++) {
        if ($section_sigil[$j] eq $sigil) {
            start_CI_loop("Section", $j, $j, $lpos);
            next CYCLE;
        }
    }
    for ($j = 0; $j<$no_chapters; $j++) {
        if ($chapter_sigil[$j] eq $sigil) {
            start_CI_loop("Chapter", $j, $j, $lpos);
            next CYCLE;
        }
    }
    inweb_error_at("don't recognise the chapter or section abbreviation $sigil",
        $path_to_template, $lpos);
    next;
}
```

This code is used in §14.

§**19.** Next, a genuine loop beginning:

⟨Deal with a Repeat command 19⟩ ≡

```
if ($command =~ m/^Repeat (Chapter|Section)$/) {
    my $from;
    my $to;
    my $lev = $1;
    if ($lev eq "Chapter") {
        $from = 0;
        $to = $no_chapters-1;
        if ($only_setting) {
            my $j;
            for ($j = 0; $j<$no_chapters; $j++) {
                if ($chapter_sigil[$j] eq $only_setting) {
                    start_CI_loop("Chapter", $j, $j, $lpos);
                    next CYCLE;
                }
            }
        }
    } elsif ($lev eq "Section") {
        my $within_chapter = heading_topmost_on_stack("Chapter");
        if ($within_chapter == -1) {
            $from = 0;
            $to = $no_sections-1;
        } else {
            $from = -1;
            my $sn;
            for ($sn = 0; $sn < $no_sections; $sn++) {
```

```
                if ($section_chap[$sn] == $within_chapter) {
                    if ($from == -1) { $from = $sn; }
                    $to = $sn;
                }
            }
        }
    } else {
        inweb_error_at("don't know how to repeat $lev: only Chapter or Section",
            $path_to_template, $lpos);
    }
    if ($from >= 0) { start_CI_loop($lev, $from, $to, $lpos); }
    next CYCLE;
}
```

This code is used in §14.

§**20.** And at the other bookend:

⟨Deal with a Repeat End command 20⟩ ≡

```
if ($command =~ m/^End (Repeat|Select)$/) {
    if ($stack_pointer <= 0) {
        inweb_error_at("stack underflow on contents template", $path_to_template, $lpos);
    }
    $repeat_stack_variable[$stack_pointer-1]++;
    if ($repeat_stack_variable[$stack_pointer-1] >=
        $repeat_stack_threshold[$stack_pointer-1]) {
        end_CI_loop();
    } else {
        $lpos = $repeat_stack_startpos[$stack_pointer-1];                    Back round loop
    }
    next CYCLE;
}
```

This code is used in §14.

§**21.** It can happen that a section loop, at least, is empty:

⟨Skip line if inside an empty loop 21⟩ ≡

```
my $rstl;
for ($rstl = $stack_pointer-1; $rstl >= 0; $rstl--) {
    if ($repeat_stack_variable[$stack_pointer-1] >=
        $repeat_stack_threshold[$stack_pointer-1]) { next CYCLE; }
}
```

This code is used in §14.

§**22.** If called with level `"Chapter"`, this returns the topmost chapter number on the stack; and similarly for `"Section"`.

```
sub heading_topmost_on_stack {
    my $level = $_[0];
    my $rstl;
    for ($rstl = $stack_pointer-1; $rstl >= 0; $rstl--) {
        if ($repeat_stack_level[$rstl] eq $level) {
            return $repeat_stack_variable[$rstl];
        }
    }
    return -1;
}
```

§**23.** This is the code for starting a loop, which stacks up the details, and similarly for ending it by popping them again:

```
sub start_CI_loop {
    my $level = $_[0];
    my $start_point = $_[1];
    my $end_point = $_[2];
    my $first_line_of_body = $_[3];
    $repeat_stack_level[$stack_pointer] = $level;
    $repeat_stack_variable[$stack_pointer] = $start_point;
    $repeat_stack_threshold[$stack_pointer] = $end_point+1;
    $repeat_stack_startpos[$stack_pointer++] = $first_line_of_body;
}
sub end_CI_loop {
    $stack_pointer--;
}
```

§**24. Variable substitutions.** We can now forget about this tiny stack machine: the one task left is to take a line from the template, and make substitutions of variables into its square-bracketed parts.

⟨Make substitutions of square-bracketed variables in line 24⟩ ≡

```
    while ($tl =~ m/^(.*?)\[\[(.*?)\]\](.*?)\s*$/) {
        my $left = $1; my $right = $3; my $subs = $2;
        if (exists($bibliographic_data{$subs})) {
            ⟨Substitute any bibliographic datum named 25⟩;
        } elsif ($subs =~ m/^(Chapter|Section|Complete) (.*)$/) {
            my $lev = $1;
            my $detail = $2;
            my $heading_number;
            if ($lev eq "Complete") {
                $heading_number = $complete_web_weave_number;
            } else {
                $heading_number = heading_topmost_on_stack($lev);
                if ($heading_number == -1) {
                    inweb_error_at("no $lev is currently selected",
                        $path_to_template, $lpos);
                }
            }
```

```
            if ($lev eq "Complete") {
                ⟨Substitute a detail about the complete PDF 26⟩;
            } elsif ($lev eq "Chapter") {
                ⟨Substitute a detail about the currently selected Chapter 27⟩;
            } else {
                ⟨Substitute a detail about the currently selected Section 28⟩;
            }
        } else {
            $subs = '<b>'.$subs.'</b>';
        }
        $tl = $left.$subs.$right;
    }
```

This code is used in §14.

§**25.** This is why, for instance, `[[Author]]` is replaced by the author's name:

⟨Substitute any bibliographic datum named 25⟩ ≡

```
    $subs = $bibliographic_data{$subs};
```

This code is used in §24.

§**26.** We store little about the complete-web-in-one-file PDF:

⟨Substitute a detail about the complete PDF 26⟩ ≡

```
    if ($detail eq "PDF Size") {
        $subs = ($pdf_size[$complete_web_weave_number]/1024)."KB";
    } elsif ($detail eq "Extent") {
        $subs = ($page_count[$complete_web_weave_number])."pp";
    } elsif ($detail eq "Leafname") {
        $subs = $complete_PDF_leafname;
    } else {
        $subs = $detail.' for complete web';
    }
```

This code is used in §24.

§**27.** And this is why [[Chapter Leafname]] turns into `$chapter_woven_pdf_leafname[$c]` for the current chapter number `$c`. Extent, PDF Size and Errors can only be determined if the chapter in question has already been woven on this run of `inweb`: hence the dashes if not.

⟨Substitute a detail about the currently selected Chapter 27⟩ ≡

```
    if ($detail eq "Title") {
        $subs = $chapter_title[$heading_number];
    } elsif ($detail eq "Purpose") {
        $subs = $chapter_rubric[$heading_number];
    } elsif ($detail eq "Leafname") {
        $subs = $chapter_woven_pdf_leafname[$heading_number];
    } elsif ($detail eq "Extent") {
        my $wtn = $chapter_weave_number[$heading_number];
        if ($wtn == -1) { $subs = "--"; } else {
            $subs = $page_count[$wtn]."pp";
        }
    } elsif ($detail eq "PDF Size") {
        my $wtn = $chapter_weave_number[$heading_number];
```

```
        if ($wtn == -1) { $subs = "--"; } else {
            $subs = ($pdf_size[$wtn]/1024)."KB";
        }
    } elsif ($detail eq "Errors") {
        my $wtn = $chapter_weave_number[$heading_number];
        if ($wtn == -1) { $subs = ""; } else {
            $subs = "";
            if ($overfull_hbox_count[$wtn] > 0) {
                $subs = $overfull_hbox_count[$wtn]." overfull hbox ";
            }
            if ($tex_error_count[$wtn] > 0) {
                $subs .= $tex_error_count[$wtn]." TeX error";
            }
        }
    } else {
        $subs = $detail.' for '.$lev.' '.$heading_number;
    }
```

This code is used in §24.

§**28.** And this, finally, is a very similar construction for Sections.

⟨Substitute a detail about the currently selected Section 28⟩ ≡

```
    if ($detail eq "Title") {
        $subs = $section_leafname[$heading_number];
        $subs =~ s/\.w$//;
    } elsif ($detail eq "Purpose") {
        $subs = $section_purpose[$heading_number];
    } elsif ($detail eq "Leafname") {
        $subs = $section_sigil[$heading_number];
        $subs =~ s/\//-/; $subs = $subs.'.pdf';
    } elsif ($detail eq "Code") {
        $subs = $section_sigil[$heading_number];
    } elsif ($detail eq "Lines") {
        $subs = $section_extent[$heading_number];
    } elsif ($detail eq "Source") {
        $subs = $section_pathname_relative_to_web[$heading_number];
    } elsif ($detail eq "Paragraphs") {
        $subs = $section_no_pars[$heading_number];
    } elsif ($detail eq "Mean") {
        my $denom = $section_no_pars[$heading_number];
        if ($denom == 0) { $denom = 1; }
        $subs = $section_extent[$heading_number]/$denom;
    } elsif ($detail eq "Extent") {
        my $wtn = $section_weave_number[$heading_number];
        if ($wtn == -1) { $subs = "--"; } else {
            $subs = $page_count[$wtn]."pp";
        }
    } elsif ($detail eq "PDF Size") {
        my $wtn = $section_weave_number[$heading_number];
        if ($wtn == -1) { $subs = "--"; } else {
            $subs = ($pdf_size[$wtn]/1024)."KB";
        }
    } elsif ($detail eq "Errors") {
```

```
        my $wtn = $section_weave_number[$heading_number];
        if ($wtn == -1) { $subs = ""; } else {
            $subs = "";
            if ($overfull_hbox_count[$wtn] > 0) {
                $subs = $overfull_hbox_count[$wtn]." overfull hbox ";
            }
            if ($tex_error_count[$wtn] > 0) {
                $subs .= $tex_error_count[$wtn]." TeX error";
            }
        }
    } else {
        $subs = $detail.' for '.$lev.' '.$heading_number;
    }
```

This code is used in §24.

The Weaver 3/weave

Purpose

To weave a portion of the code into instructions for TeX.

§**1. The Master Weaver.** Here's what has happened so far, on a `-weave` run of `inweb`: on any other sort of run, of course, we would never be in this section of code. The web was read completely into memory, and then fully parsed, with all of the arrays and hashes populated. A request was then made either to swarm a mass of individual weaves, or to make just a single weave, with the target in each case being identified by its sigil. A further decoding layer then translated each sigil into rather more basic details of what to weave and where to put the result: and so we arrive at the front door of the routine `weave_source` below.

```
sub weave_source {
    my $filename = $_[0];
    my $with_cover_sheet = $_[1];                    Should we add a cover sheet at the front?
    my $weave_section_match = $_[2];                 Regular expression which sigils for the lines must match
    open(WEAVEOUT, ">".$filename) or die "inweb: can't open weave file '$filename' for output";
    print WEAVEOUT "% Weave ", $filename, " generated by ", $INWEB_BUILD, "\n";
    ⟨Incorporate suitable TEX macro definitions into the woven output 3⟩;
    if ($with_cover_sheet == 1) { weave_cover_sheet(); }
    ⟨Start the weaver with a clean slate 2⟩;
    OUTLOOP:
    for ($i=0; $i<$no_lines; $i++) {
        ⟨Skip material from any sections which are not part of this target 4⟩;
        $lines_woven++;                              In principle, then, what remains is in the target; but...
        ⟨Material between ...and so on... markers is not visible 5⟩;
        ⟨Grammar and index entries are collated elsewhere, not woven in situ 6⟩;
        ⟨Respond to any commands aimed at the weaver, and otherwise skip commands 7⟩;
        Some of the more baroque front matter of a section...
        ⟨Weave the Purpose marker as a little heading 9⟩;
        ⟨If we need to work in a section table of contents and this is a blank line, do it now 10⟩;
        ⟨Deal with the Interface passage 11⟩;
        ⟨Weave the Definitions marker as a little heading 12⟩;
        ⟨Weave the section bar as a horizontal rule 13⟩;
        The crucial junction point between modes...
        ⟨Deal with the marker for the start of a new paragraph, section or chapter 32⟩;
        With all exotica dealt with, we now just have material to weave verbatim...
        my $matter = $line_text[$i];
        if ($line_is_comment[$i] == 1) ⟨Weave verbatim matter in commentary style 14⟩
        else ⟨Weave verbatim matter in code style 19⟩;
    }
    ⟨Complete any started but not-fully-woven paragraph 36⟩;
    print WEAVEOUT "% End of weave: ", $lines_woven, " lines from ", $no_lines, "\n";
    print WEAVEOUT '\end', "\n";
    close(WEAVEOUT);
}
```

§**2.** We can now begin on a clean page, by initialising the state of the weaver. It's convenient for these to be global variables since the weaver is not recursively called, and it avoids some nuisance over scope caused by the braces implicit in the `inweb` macro below:

⟨Start the weaver with a clean slate 2⟩ ≡

```
$within_TeX_beginlines = 0;                    Currently setting copy between \beginlines and \endlines?
$weaving_suspended = 0;
$interface_table_pending = 0;
$functions_in_par = ""; $structs_in_par = "";
$current_macro_definition = "";
$next_heading_without_vertical_skip = 0;
$show_section_toc_soon = 0;                    Is a table of contents for the section imminent?
$horizontal_rule_just_drawn = 0;
$chaptermark = ""; $sectionmark = "";
begin_making_pdf_links();                      Be ready to set PDF hyperlinks from now on
$lines_woven = 0;                              Number of lines in the target to be woven
```

This code is used in §1.

§**3. The TEX macros.** We don't use TEX's `\input` mechanism for macros because it is so prone to failures when searching directories (especially those with spaces in the names) and then locking TEX into a repeated prompt for help from `stdin` which is rather hard to escape from.

Instead we paste the entire text of our macros file into the woven TEX:

⟨Incorporate suitable TEX macro definitions into the woven output 3⟩ ≡

```
my $ml;
open(MACROS, $path_to_inweb_setting.'inweb/Materials/inweb-macros.tex')
    or die "inweb: can't open file of TeX macros";
while ($ml = <MACROS>) {
    $ml =~ m/^(.*)\s*$/; $ml = $1;
    print WEAVEOUT $ml, "\n";
}
close MACROS;
```

This code is used in §1.

§**4. Reasons to skip things.** We skip any material in files not chosen at the command line (or by the swarmer) for weave output:

⟨Skip material from any sections which are not part of this target 4⟩ ≡

```
if ($weave_section_match ne "") {
    if (not ($section_sigil[$line_sec[$i]] =~ m/$weave_section_match/)) { next OUTLOOP; }
}
```

This code is used in §1.

§**5.** We skip material between "...and so on..." markers as being even more tedious than the rest of the program:

⟨Material between ...and so on... markers is not visible 5⟩ ≡
```
if ($line_category[$i] == $TOGGLE_WEAVING_LCAT) {
    if ($weaving_suspended == 0) {
        print WEAVEOUT "\\smallskip\\par\\noindent";
        print WEAVEOUT "{\\ttninepoint\\it ...and so on...}\\smallskip\n";
        $weaving_suspended = 1;
        next OUTLOOP;
    }
    $weaving_suspended = 0;
    next OUTLOOP;
}
if ($weaving_suspended == 1) { next OUTLOOP; }
```
This code is used in §1.

§**6.** And we skip some material used only for compiling tables:

⟨Grammar and index entries are collated elsewhere, not woven in situ 6⟩ ≡
```
if ($line_category[$i] == $GRAMMAR_LCAT) { next OUTLOOP; }
if ($line_category[$i] == $GRAMMAR_BODY_LCAT) { next OUTLOOP; }
if ($line_category[$i] == $INDEX_ENTRY_LCAT) { next OUTLOOP; }
```
This code is used in §1.

§**7.** And lastly we ignore commands, or act on them if they happen to be aimed at us; but we don't weave them into the output, that's for sure.

⟨Respond to any commands aimed at the weaver, and otherwise skip commands 7⟩ ≡
```
if ($line_category[$i] == $COMMAND_LCAT) {
    my $argument = $line_operand_2[$i];
    if ($line_operand[$i] == $PAGEBREAK_CMD) {
        print WEAVEOUT "\\vfill\\eject\n";
    }
    if ($line_operand[$i] == $BNF_GRAMMAR_CMD) {
        weave_bnf_grammar($argument);
        print WEAVEOUT "\\smallbreak\n";
        print WEAVEOUT "\\hrule\\smallbreak\n";
    }
    if ($line_operand[$i] == $THEMATIC_INDEX_CMD) {
        weave_thematic_index($argument);
        print WEAVEOUT "\\smallbreak\n";
        print WEAVEOUT "\\hrule\\smallbreak\n";
    }
    if ($line_operand[$i] == $FIGURE_CMD) {
        ⟨Weave a figure 8⟩;
    }
    Otherwise assume it was a tangler command, and ignore it here
    next OUTLOOP;
}
```
This code is used in §1.

§**8.** TEX itself has an almost defiant lack of support for anything pictorial, which is one reason it didn't live up to its hope of being the definitive basis for typography; even today the loose confederation of TEX-like programs and extensions lack standard approaches. Here we're going to use `pdftex` features, having nothing better. All we're trying for is to insert a picture, scaled to a given width, into the text at the current position.

⟨Weave a figure 8⟩ ≡
```
my $figname = $argument;
my $width = "";
if ($figname =~ m/^(\d+)cm\: (.*)$/) {
    $figname = $2;
    $width = " width ".$1."cm";
}
print WEAVEOUT "\\pdfximage".$width."{../Figures/".$figname, "}\n";
print WEAVEOUT "\\smallskip\\noindent",
    "\\hbox to\\hsize{\\hfill\\pdfrefximage \\pdflastximage\\hfill}",
    "\\smallskip\n";
```
This code is used in §7.

§**9. Headings.** The purpose is set with a little heading. Its operand is that part of the purpose-text which is on the opening line; the rest follows on subsequent lines until the next blank.

⟨Weave the Purpose marker as a little heading 9⟩ ≡
```
if ($line_category[$i] == $PURPOSE_LCAT) {
    print WEAVEOUT "\\smallskip\\par\\noindent{\\it Purpose}\\smallskip\\noindent\n";
    print WEAVEOUT $line_operand[$i], "\n";
    $show_section_toc_soon = 1;
    $horizontal_rule_just_drawn = 0;
    next OUTLOOP;
}
```
This code is used in §1.

§**10.** This normally appears just after the Purpose subheading:

⟨If we need to work in a section table of contents and this is a blank line, do it now 10⟩ ≡
```
if (($show_section_toc_soon == 1) && ($line_text[$i] =~ m/^\s*$/)) {
    print WEAVEOUT "\\medskip";
    $show_section_toc_soon = 0;
    if ($section_toc[$line_sec[$i]] ne "") {
        print WEAVEOUT "\\smallskip\\hrule\\smallskip\\par\\noindent{\\usagefont ",
            $section_toc[$line_sec[$i]], "}\\par\\medskip\\hrule\\bigskip\n";
        $horizontal_rule_just_drawn = 1;
    } else {
        $horizontal_rule_just_drawn = 0;
    }
}
```
This code is used in §1.

§**11.** After which we have the Interface, except that this is skipped entirely, as far as weaving is concerned, unless (a) there's a body to it, and (b) we are in C-for-Inform mode, in which case the body is ignored anyway but a table of I6 template invocations of the section's functions is set instead:

⟨Deal with the Interface passage 11⟩ ≡

```
if ($line_category[$i] == $INTERFACE_LCAT) {
    $interface_table_pending = 1;
    next OUTLOOP;
}
if ($line_category[$i] == $INTERFACE_BODY_LCAT) {
    if ($interface_table_pending) {
        $interface_table_pending = 0;
        if ($web_language == $C_FOR_INFORM_LANGUAGE) {
            $horizontal_rule_just_drawn = 0;
            weave_interface_table_for_section($line_sec[$i]);
        }
    }
    next OUTLOOP;
}
```

This code is used in §1.

§**12.** And another little heading...

⟨Weave the Definitions marker as a little heading 12⟩ ≡

```
if ($line_category[$i] == $DEFINITIONS_LCAT) {
    print WEAVEOUT "\\smallskip\\par\\noindent{\\it Definitions}\\smallskip\\noindent\n";
    $next_heading_without_vertical_skip = 1;
    $horizontal_rule_just_drawn = 0;
    next OUTLOOP;
}
```

This code is used in §1.

§**13.** ...with the section bar to follow. The bar line completes any half-finished paragraph and is set as a horizontal rule:

⟨Weave the section bar as a horizontal rule 13⟩ ≡

```
if ($line_category[$i] == $BAR_LCAT) {
    ⟨Complete any started but not-fully-woven paragraph 36⟩;
    $functions_in_par = "";
    $structs_in_par = "";
    $current_macro_definition = "";
    $within_TeX_beginlines = 0;
    $next_heading_without_vertical_skip = 1;
    if ($horizontal_rule_just_drawn == 0) {
        print WEAVEOUT "\\par\\medskip\\noindent\\hrule\\medskip\\noindent\n";
    }
    next OUTLOOP;
}
```

This code is used in §1.

§14. Commentary matter. Typographically this is a fairly simple business, since the commentary is already by definition written in TEX format: it's almost the case that we only have to transcribe it. But not quite! This is where we implement the convenient additions `inweb` makes to TEX syntax.

⟨Weave verbatim matter in commentary style 14⟩ ≡

```
⟨Weave displayed source in its own special style 15⟩;
⟨Weave a blank line as a thin vertical skip and paragraph break 16⟩;
⟨Weave bracketed list indications at start of line into indentation 17⟩;
⟨Weave tabbed code material as a new indented paragraph 18⟩;
print WEAVEOUT $matter, "\n";
next OUTLOOP;
```

This code is used in §1.

§15. Displayed source is the material marked with >> arrows in column 1.

⟨Weave displayed source in its own special style 15⟩ ≡

```
if ($line_category[$i] == $SOURCE_DISPLAY_LCAT) {
    print WEAVEOUT "\\quotesource{", $line_operand[$i], "}\n";
    next OUTLOOP;
}
```

This code is used in §14.

§16. Our style is to use paragraphs without initial-line indentation, so we add a vertical skip between them to show the division more clearly.

⟨Weave a blank line as a thin vertical skip and paragraph break 16⟩ ≡

```
if ($matter =~ m/^\s*$/) {
    print WEAVEOUT "\\smallskip\\par\\noindent%\n";
    next OUTLOOP;
}
```

This code is used in §14.

§17. Here our extension is simply to provide a tidier way to use TEX's standard `\item` and `\itemitem` macros for indented list items.

⟨Weave bracketed list indications at start of line into indentation 17⟩ ≡

```
if ($matter =~ m/^\(\.\.\.\)\s+(.*?)$/) {                    continue single
    $matter = '\item{}'.$1;
} elsif ($matter =~ m/^\(-\.\.\.\)\s+(.*?)$/) {              continue double
    $matter = '\itemitem{}'.$1;
} elsif ($matter =~ m/^\(([a-z0-9A-Z\.]+)\)\s+(.*?)$/) {     begin single
    $matter = '\item{('.$1.')}'.$2;
} elsif ($matter =~ m/^\(-([a-z0-9A-Z\.]+)\)\s+(.*?)$/) {    begin double
    $matter = '\itemitem{('.$1.')}'.$2;
}
```

This code is used in §14.

§**18.** Finally, matter encased in vertical strokes one tab stop in from column 1 in the source is set indented in code style.

⟨Weave tabbed code material as a new indented paragraph 18⟩ ≡
```
if ($matter =~ m/^\t\|/) {
    $matter = '\par\noindent\qquad '.$matter;
}
```
This code is used in §14.

§**19. Code-like matter.** Even though `inweb`'s approach, unlike `CWEB`'s, is to respect the layout of the original, this is still quite typographically complex: commentary and macro usage is rendered differently.

⟨Weave verbatim matter in code style 19⟩ ≡
```
⟨Enter beginlines/endlines mode if necessary 20⟩;
⟨Weave a blank line as a thin vertical skip 21⟩;
my $tab_stops_of_indentation = 0;
⟨Convert leading space in line matter to a number of tab stops 22⟩;
⟨Weave a suitable horizontal advance for that many tab stops 23⟩;
my $concluding_comment = "";
⟨Extract any comment matter ending the line to be set in italic 24⟩;
⟨Encase the code matter within vertical strokes 25⟩;
⟨Give constant definition lines slightly fancier openings 26⟩;
⟨Detect any structure or function definitions being woven in this paragraph 27⟩;
⟨Typeset the CWEB-style macros with cute highlighting and PDF links 28⟩;
⟨Insert continuation line breaks to cope with very long lines 29⟩;
print WEAVEOUT $matter, $concluding_comment, "\n";
next OUTLOOP;
```
This code is used in §1.

§**20.** Code is typeset between the `\beginlines` and `\endlines` macros in TEX, so:

⟨Enter beginlines/endlines mode if necessary 20⟩ ≡
```
if ($within_TeX_beginlines == 0) {
    print WEAVEOUT "\\beginlines\n"; $within_TeX_beginlines = 1;
}
```
This code is used in §19.

§**21.** A blank line is typeset as a thin vertical skip (no TEX paragraph break is needed):

⟨Weave a blank line as a thin vertical skip 21⟩ ≡
```
if ($matter =~ m/^\s*$/) {
    print WEAVEOUT "\\smallskip\n";
    next OUTLOOP;
}
```
This code is used in §19.

§**22.** Examine the white space at the start of the code line, and count the number of tab steps of indentation, rating 1 tab = 4 spaces:

⟨Convert leading space in line matter to a number of tab stops 22⟩ ≡

```
my $spaces_in = 0;
while ($matter =~ m/^(\s)(.*)$/) {
    $matter = $2;
    $whitespace = $1;
    if ($whitespace eq "\t") {
        $spaces_in = 0;
        $tab_stops_of_indentation++;
    } else {
        $spaces_in++;
        if ($spaces_in == 4) {
            $tab_stops_of_indentation++;
            $spaces_in = 0;
        }
    }
}
```

This code is used in §19.

§**23.** We actually use horizontal spaces rather than risk using TEX's messy alignment system:

⟨Weave a suitable horizontal advance for that many tab stops 23⟩ ≡

```
my $i;
for ($i=0; $i<$tab_stops_of_indentation; $i++) {
    print WEAVEOUT "\\qquad";
}
```

This code is used in §19.

§**24.** Comments which run to the end of a line are set in italic type. If the only item on their lines, they are presented at the code tab stop; otherwise, they are set flush right.

⟨Extract any comment matter ending the line to be set in italic 24⟩ ≡

```
if (line_ends_with_comment($matter)) {
    if ($part_before_comment eq "") {
        $matter = $part_before_comment; my $commentary = $part_within_comment;
        $concluding_comment = "{\\ttninepoint\\it ".$commentary."}";
    } else {
        $matter = $part_before_comment; my $commentary = $part_within_comment;
        if ($commentary =~ m/^C\d+\S+$/) {
            $commentary = "Test with |".$commentary.".txt|";
        }
        $concluding_comment = "\\hfill\\quad {\\ttninepoint\\it ".$commentary."}";
    }
}
```

This code is used in §19.

§**25.** Code is typeset by TEX within vertical strokes; these switch a sort of typewriter-type verbatim mode on and off. To get an actual stroke, we must escape from code mode, escape it using a backslash `|`, then re-enter code mode once again:

⟨Encase the code matter within vertical strokes 25⟩ ≡
```
$matter =~ s/\|/\|\\\|\|/g;
$matter = '|'.$matter.'|';
```
This code is used in §19.

§**26.** Set the `@d` definition escape very slightly more fancily (remembering that we are now encased in verticals):

⟨Give constant definition lines slightly fancier openings 26⟩ ≡
```
$matter =~ s/^\|\@d /{\\ninebf define}| /;
```
This code is used in §19.

§**27.** We note any structure typedefs, and also any functions which are called from outside this section, whose names we typeset in red. (We do this so that the endnotes can be added at the foot of the paragraph.)

⟨Detect any structure or function definitions being woven in this paragraph 27⟩ ≡
```
if ($matter =~ m/^\|\s*typedef\s+struct\s+(.*?)\s+\{/) {
    $structs_in_par .= $1.",";
}
if ($matter =~ m/^\|\/(\*+)\/\s*(.*?\**)(\S+?)\((.*)$/) {
    my $fstars = $1;
    my $ftype = $2;
    my $fname = $3;
    my $frest = $4;
    $matter = '|'.$ftype.'|\pdfliteral direct{0 1 1 0 k}|'.$fname.'|\special{PDF:0 g}|'.
        '('.$frest;
    $functions_in_par .= $fname.",";
}
if (($tab_stops_of_indentation == 0) &&
    ($matter =~ m/^\|(\S.*?\**)(\S+?)\((.*)$/)) {
    my $ftype = $1;
    my $fname = $2;
    my $frest = $3;
    my $unamended = $fname;
    $fname =~ s/::/__/g;
    if (exists($functions_line{$fname})) {
        $matter = '|'.$ftype.'|\pdfliteral direct{0 1 1 0 k}|'.$unamended.
            '|\special{PDF:0 g}|'.'('.$frest;
        $functions_in_par .= $fname.",";
    }
}
```
This code is used in §19.

§**28.** Any usage of angle-macros is highlighted in several cute ways: first, we make use of colour and we drop in the paragraph number of the definition of the macro in small type –

⟨Typeset the CWEB-style macros with cute highlighting and PDF links 28⟩ ≡
```
while ($matter =~ m/^(.*?)\@\<(.*?)\@\>(.*)$/) {
    my $left = $1;
    my $right = $3;
    my $href = $2.' {\sevenss '.$cweb_macros_paragraph_number{$2}.'}';
    my $angled_sname = $2;
    my $xrefcol = '\pdfliteral direct{1 1 0 0 k}';
    my $blackcol = '\special{PDF:0 g}';
    my $tweaked = '$\langle${\xreffont'.$xrefcol.$href.'}'.$blackcol.'$\rangle$';
    if ($right =~ m/^\s*\=(.*)$/) ⟨This is where the angle-macro is defined 30⟩
    else ⟨This is a reference to an angle-macro, not its definition 31⟩;
    $matter = $left.'|'.$tweaked.'|'.$right;
}
```
This code is used in §19.

§**29.** Overlong lines of code are liable to cause overfull hbox errors, the bane of all TEX users.

⟨Insert continuation line breaks to cope with very long lines 29⟩ ≡
```
my $count = 0;
my $code_matter = 0;
my $done = "";
my $number_of_splits = 0;
my $last_char = "";
while ($matter =~ m/^(.)(.*?)$/) {
    my $next_char = $1; $matter = $2;
    if (($next_char eq "|") && ($last_char ne "\\")) { $code_matter = 1 - $code_matter; }
    elsif ($code_matter == 1) {
        $count++;
        if ($count > 99) {
            $done = $done."| \n\\quad ...| "; $number_of_splits++; $count = 10;
        }
    }
    $done = $done.$next_char;
    $last_char = $next_char;
}
$matter = $done;
```
This code is used in §19.

§**30.** We mark this as a hyperlink destination (using `pdftex` extensions again):

⟨This is where the angle-macro is defined 30⟩ ≡
```
$right = $1;
$tweaked = '\pdfdest num '.link_for_par_name($angled_sname).' fit '.$tweaked;
$tweaked = $tweaked.' $\equiv$ ';
$current_macro_definition = $angled_sname;
```
This code is used in §28.

§**31.** And this is a link:

⟨This is a reference to an angle-macro, not its definition 31⟩ ≡

```
my $new_p = $line_paragraph_number[$i];
if (not($angled_paragraph_usage{$angled_sname} =~ m/$new_p,$/)) {
    $angled_paragraph_usage{$angled_sname} .= $new_p.',';
}
$tweaked =
    '\pdfstartlink attr{/C [0.9 0 0] /Border [0 0 0]} '.
    'goto num '.link_for_par_name($angled_sname).' '.
    $tweaked.'\pdfendlink';
```

This code is used in §28.

§32. How paragraphs begin.

⟨Deal with the marker for the start of a new paragraph, section or chapter 32⟩ ≡

```
if ($line_category[$i] == $PARAGRAPH_START_LCAT) {
    ⟨Complete any started but not-fully-woven paragraph 36⟩;
    ⟨If this is a paragraph break forced onto a new page, then throw a page 33⟩;
    my $weight = $line_operand[$i];
    my $para_title = $line_operand_2[$i];
    my $new_chap_num = $section_chap[$line_sec[$i]];
    my $secnum = $line_sec[$i];
    my $parnum = $line_paragraph_number[$i];
    my $ornament = $line_paragraph_ornament[$i];
    my $mark = "";
    ⟨Work out the next mark to place into the TEX vertical list 34⟩;
    $functions_in_par = "";
    $structs_in_par = "";
    $current_macro_definition = "";
    my $TeX_macro = "";
    ⟨Choose which TEX macro to use in order to typeset the new paragraph heading 35⟩;
    print WEAVEOUT "\\".$TeX_macro, "{", $parnum, "}{", $para_title, "}{",
        $mark, "}{", $ornament, "}{", $section_sigil[$secnum], "}%\n";
    There's quite likely ordinary text on the line following the paragraph
    start indication, too, so we need to weave this out:
    if ($line_text[$i] ne "") { print WEAVEOUT $line_text[$i], "\n"; }
    Chapter headings get a chapter title page, or possibly pages, too:
    if ($weight == 3) {
        my $sigil = $chapter_sigil[$section_chap[$line_sec[$i]]];
        print WEAVEOUT $chapter_rubric{$sigil}, "\\medskip\n";
        my $sn;
        for ($sn=0; $sn<$no_sections; $sn++) {
            if (not($section_sigil[$sn] =~ m/^$sigil\//)) { next; }
            print WEAVEOUT "\\smallskip\\noindent |", $section_sigil[$sn], "|: ",
                "{\\it ", $section_leafname[$sn], "}\\quad\n";
            print WEAVEOUT $section_purpose[$sn];
        }
    }
    And that completes the new paragraph opening.
    next OUTLOOP;
}
```

This code is used in §1.

§33. A few paragraphs are marked `@pp` and forced to begin on a new page:

⟨If this is a paragraph break forced onto a new page, then throw a page 33⟩ ≡

```
if ($line_starts_paragraph_on_new_page[$i]) {
    print WEAVEOUT "\\vfill\\eject\n";
}
```

This code is used in §32.

§**34.** "Marks" are the vile contrivance by which TEX produces running heads on pages which follow the material on those pages: so that the running head for a page can show the paragraph sigil for the material which tops it, for instance.

The ornament has to be set in math mode, even in the mark. § and ¶ only work in math mode because they abbreviate characters found in math fonts but not regular ones, in TEX's slightly peculiar font encoding system.

define $DEBUG_MARKING 0

⟨Work out the next mark to place into the TEX vertical list 34⟩ ≡

```
if ($weight == 3) { $chaptermark = $para_title; $sectionmark = ""; }
if ($weight == 2) {
    $sectionmark = $para_title;
    if ($section_sigil[$secnum] ne "") { $chaptermark = $section_sigil[$secnum]; }
    if ($chaptermark ne "") { $sectionmark = " - ".$sectionmark; }
}
$mark = $chaptermark.$sectionmark."\\quad\$".$ornament."\$".$parnum;
if ($DEBUG_MARKING == 1) {
    print "Start par $i: weight $weight, title <$para_title>, ",
        "chap $new_chap_num, sec $secnum, par $parnum, ornament $ornament\n";
    print "Mark: $mark\n";
}
```

This code is used in §32.

§**35.** We want to have different heading styles for different weights, and TEX is horrible at using macro parameters as function arguments, so we don't want to pass the weight that way. Instead we use

```
\weavesection
\weavesections
\weavesectionss
\weavesectionsss
```

where the weight is the number of terminal ss, 0 to 3. (TEX macros, lamentably, are not allowed digits in their name.) In the cases 0 and 1, we also have variants `\nsweavesection` and `\nsweavesections` which are the same, but with the initial vertical spacing removed; these allow us to prevent unsightly excess white space in certain configurations of a section.

⟨Choose which TEX macro to use in order to typeset the new paragraph heading 35⟩ ≡

```
my $j;
$TeX_macro = "weavesection";
for ($j=0; $j<$weight; $j++) { $TeX_macro = $TeX_macro."s"; }
if (($next_heading_without_vertical_skip == 1) && ($weight < 2)) {
    $next_heading_without_vertical_skip = 0;
    $TeX_macro = "ns".$TeX_macro;
}
if ($weight == 3) {                                    a chapter heading: note the inch-high sigil...
    my $brief_title = $para_title;
    $brief_title =~ s/^.*?\: //;
    $para_title = "{\\sinchhigh ".$chapter_sigil[$section_chap[$line_sec[$i]]].
        "}\\quad ".$brief_title;
}
```

This code is used in §32.

§**36. How paragraphs end.** At the end of a paragraph, on the other hand, we do this:

⟨Complete any started but not-fully-woven paragraph 36⟩ ≡

```
if ($within_TeX_beginlines > 0) {
    print WEAVEOUT '\endlines', "\n"; $within_TeX_beginlines = 0;
}
show_endnotes_on_previous_paragraph($functions_in_par, $structs_in_par,
    $current_macro_definition);
```

This code is used in §1,13,32,1,13,32.

§**37.** The endnotes describe function calls from far away, or unexpected structure usage, or how CWEB-style code substitutions were made.

```
sub show_endnotes_on_previous_paragraph {
    my $functions_in_par = $_[0];
    my $structs_in_par = $_[1];
    my $current_macro_definition = $_[2];
    if ($current_macro_definition ne "")
        ⟨Weave endnote saying which other paragraphs use this one 38⟩
    else {
        my %fnames_done = ();
        while ($functions_in_par =~ m/^(\S+?)\,(.*)$/) {
            my $fname = $1;
            $functions_in_par = $2;
            if ($fnames_done{$fname}++ > 0) { next; }
            if ($fname eq "isdigit") { next; }
            ⟨Weave endnote saying where this function is called from 39⟩;
        }
        while ($structs_in_par =~ m/^(\S+?)\,(.*)$/) {
            my $sname = $1;
            my $sbilling;
            $structs_in_par = $2;
            if ($structure_ownership_summary{$sname} ne "") {
                $sbilling = $structure_ownership_summary{$sname};
                $sbilling =~ s/ /, /g;
                $sbilling =~ s/, $//;
                $sbilling =~ s/, (\S+)$/ and $1/;
                $sbilling = "shared with ".$sbilling;
            } else {
                $sbilling = "private to this section";
            }
            $sname =~ s/_/\\_/g;
            print WEAVEOUT "\\par\\noindent";
            print WEAVEOUT "\\penalty10000\n";
            $sname =~ s/\#/\\#/g;
            print WEAVEOUT "{\\usagefont The structure $sname is $sbilling.}\n";
        }
        print WEAVEOUT "\\smallskip\n";
    }
}
```

§**38.** We try hard to prevent a page break between paragraph and endnote (hence the high `\penalty` value). Sometimes this succeeds.

⟨Weave endnote saying which other paragraphs use this one 38⟩ ≡

```
my $used_in = $angled_paragraph_usage{$current_macro_definition};
$used_in =~ s/\,$//;
print WEAVEOUT "\\penalty1000\n";
print WEAVEOUT "\\noindent{\\usagefont This code is used in \$\\S\$",
    $used_in, ".}\\smallskip\n";
```

This code is used in §37.

§**39.** And similarly:

⟨Weave endnote saying where this function is called from 39⟩ ≡

```
my $scope = $functions_declared_scope{$fname};
my @usages;
my $no_sections_using = 0;
⟨Work out which sections to mention calls from 40⟩;
my $fname_with_underscores_escaped = $fname;
$fname_with_underscores_escaped =~ s/__/::/g;
$fname_with_underscores_escaped =~ s/_/\\_/g;
print WEAVEOUT "\\par\\noindent";
print WEAVEOUT "\\penalty10000\n";
print WEAVEOUT "{\\usagefont The function $fname_with_underscores_escaped is";
my $clause_begins = 0;
if ($scope eq "*****") {
    print WEAVEOUT " where execution begins"; $clause_begins = 1;
}
if ($scope eq "****") {
    print WEAVEOUT " invoked by a command in a |.i6t| template file"; $clause_begins = 1;
}
if (($clause_begins == 1) || (($scope eq "***") || ($scope eq "**"))) {
    if ($no_sections_using > 0) {
        if ($clause_begins) { print WEAVEOUT " and"; }
        print WEAVEOUT " called from ";
        my $x;
        for ($x=0; $x<$no_sections; $x++) {
            if ($usages[$x] == 1) {
                print WEAVEOUT $section_sigil[$x];
                if ($no_sections_using > 2) { print WEAVEOUT ", "; }
                if ($no_sections_using == 2) { print WEAVEOUT " and "; }
                $no_sections_using--;
            }
        }
    }
}
print WEAVEOUT ".}\n";
```

This code is used in §37.

§**40.** Loop through the concise string holding this information:

⟨Work out which sections to mention calls from 40⟩ ≡

```
my $x;
for ($x=0; $x<$no_sections; $x++) { $usages[$x] = 0; }
my $cp = $functions_usage_concisely_described{$fname};
while ($cp =~ m/^\:(\d+)(.*?)$/) { $usages[eval($1)] = 1; $cp = $2; }
for ($x=0; $x<$no_sections; $x++) { if ($usages[$x] == 1) { $no_sections_using++; } }
```

This code is used in §39.

§**41. The cover sheet.** This is added to the front of larger PDFs; whole chapters and the complete work. People will probably want to customise this, so it's implemented as a TEX file which we substitute bibliographic data into.

```
sub weave_cover_sheet {
    my $cover_sheet = $path_to_inweb_setting.'inweb/Materials/cover-sheet.tex';
    if (exists ($bibliographic_data{"Cover Sheet"})) {
        $cover_sheet = $web_setting.'Materials/'.$bibliographic_data{"Cover Sheet"};
    }
    $bibliographic_data{"Capitalized Title"} = $bibliographic_data{"Title"};
    $bibliographic_data{"Capitalized Title"} =~ tr/a-z/A-Z/;
    $bibliographic_data{"Booklet Title"} = $booklet_title;
    weave_cover_from($cover_sheet);
}
```

§**42.** Note that the substitution `[[Cover Sheet]]` embeds the whole default cover sheet; this is so that we can just add a continuation of that, if we want to.

```
sub weave_cover_from {
    my $cs_filename = $_[0];
    local *COVER;
    open(COVER, $cs_filename) or die "inweb: cannot find cover sheet file";
    my $csl;
    while ($csl = <COVER>) {
        $csl =~ m/^(.*?)\s*$/; $csl = $1;
        if ($csl =~ m/^\%/) { next; }                    a TeX comment begins with a percent sign
        while ($csl =~ m/^(.*?)\[\[(.*?)\]\](.*)$/) {
            print WEAVEOUT $1; my $expander = $2; $csl = $3;
            if ($expander eq "Cover Sheet") {
                my $insert_filename =
                    $path_to_inweb_setting.'inweb/Materials/cover-sheet.tex';
                if ($insert_filename ne $cs_filename) {
                    weave_cover_from($insert_filename);
                }
            } elsif (exists($bibliographic_data{$expander})) {
                print WEAVEOUT $bibliographic_data{$expander};
            } else {
                print WEAVEOUT $expander;
            }
        }
        print WEAVEOUT $csl, "\n";
    }
```

```
    close (COVER);
}
```

§**43. Table of I6 template interpreter invocations.** This is only used in C-for-Inform mode, and shows four-star functions – the ones called by the I6T interpreter.

```
sub weave_interface_table_for_section {
    my $sect_no = $_[0];
    my $j;
    my $red_text = "\|\\pdfliteral direct\{0 1 1 0 k\}\|";
    my $blue_text = "\|\\pdfliteral direct\{1 1 0 0 k\}\|";
    my $black_text = "\|\\special{PDF:0 g}\|";
    my $fname;
    my %functions_in_order = ();                          a device used to sort functions in definition order
    foreach $fname (keys %functions_line) {
        $functions_in_order{sprintf("%07d", $functions_line{$fname})} = $fname;
    }
    my $heading_made = 0;
    foreach $fnamed (sort keys %functions_in_order) {
        $fname = $functions_in_order{$fnamed};
        my $j = $functions_line{$fname};
        if ($line_sec[$j] != $sect_no) { next; }
        if ($functions_declared_scope{$fname} ne "****") { next; }
        if ($heading_made == 0) {
            $heading_made = 1;
            print WEAVEOUT "\\bigbreak\\par\\noindent",
                "{\\it Template interpreter commands}\\smallskip\n";
        }
        print WEAVEOUT "\\par\\noindent\n";
        print WEAVEOUT "{\\sevenss", $line_paragraph_number[$j], "}|  |\n";
        $spc = $functions_arglist{$fname};
        $spc =~ s/\,\s*/\,\n\t\t/g;
        $access = 'callv';
        if ($spc =~ m/FILE \*/) { $access = 'call'; }
        if ($fname =~ m/^(.*)_array$/) { $fname = $1; $access = 'array'; }
        if ($fname =~ m/^(.*)_routine$/) { $fname = $1; $access = 'routine'; }
        $fname =~ s/__/::/g;
        print WEAVEOUT '|{-', $access, ':', $blue_text, $fname, $black_text, "}|\n";
    }
    if ($heading_made == 1) {
        print WEAVEOUT "\\bigbreak\\noindent\n";
    }
}
```

§**44. Thematic Index.** Produced in response to explicit weaver commands (and otherwise never visible, so in a web which doesn't use indexing, this will never have any effect).

```
sub weave_thematic_index {
    my $wanted= $_[0];
    foreach $subject (sort keys %thematic_indices) {
        my $chunk;
        if (($wanted ne "Remainder") && ($wanted ne $subject)) { next; }
        if ($index_subject_done{$subject}) { next; }
        $index_subject_done{$subject} = 1;
        print WEAVEOUT "{\\bf{", $subject, "}}\\par\n\\smallbreak\\hrule\\smallbreak\n";
        print WEAVEOUT ;
        $chunk = $thematic_indices{$subject};
        while ($chunk =~ m/^(.*?)\.\.\.(\d+)\|(.*)$/) {
            my $entry = $1;
            my $at = $2;
            $chunk = $3;
            print WEAVEOUT "\\line{", $entry,
                "\\leaders\\hbox to 1em{\\hss.\\hss}\\hfill {\\xreffont ",
                $line_csn[eval($at)], "}}\n";
        }
    }
}
```

§**45. PDF links.** Hypertext links within a PDF document are now a pretty well-established part of the PDF specification: we don't set any that point outside it. TEX of course cannot generate such details, but `pdftex` and other more modern forms can.

```
sub begin_making_pdf_links {
    %link_number_for_par_name = ();
    $no_par_name_links = 1;
}
sub link_for_par_name {
    my $par_name = $_[0];
    if (not(exists $link_number_for_par_name{$par_name})) {
        $link_number_for_par_name{$par_name} = $no_par_name_links++;
    }
    return $link_number_for_par_name{$par_name};
}
```

Backus-Naur Form 3/bnf

Purpose

To weave any collated BNF grammar from the web into a nicely typeset form.

§**1. Parsing BNF.** See the manual for more on what these grammars are, and how they're notated in the web. They have no function in the compiled code, and are purely for documentation. It's a diversion rather than the main work of the weaver, but it may as well stay in `inweb`, though it won't be useful for many webs.

There are shameless Inform-specific hacks in this section, but they're not likely to do any harm.

When we read and parsed the web, we extracted the BNF grammar lines into an array. But they were otherwise raw, and we will have to manipulate them quite a bit to turn them into TEX.

First, a little bit of jargon. The tokens we represent in angle brackets are called "nonterminals", and the grammar lines which provide ways they can be made up are called "productions". Our grammar contains only lines in which the left hand side is a single nonterminal.

```
sub parse_bnf_grammar {
    ⟨Go through the raw lines of BNF grammar, stringing together, and spotting nonterminals 2⟩;
    ⟨Initialise the nonterminals discovered 6⟩;
    ⟨Work out which nonterminals depend on which other ones 7⟩;
    ⟨Elect certain nonterminals as being fundamental 8⟩;
    ⟨Make the fundamental nonterminals the roots of trees of those depending on them 9⟩;
    ⟨Any nonterminals remaining are placed at the top level 10⟩;
}
```

§**2.** Our basic plan is to use the hash `$productions{$nt}` to accumulate all productions which have the nonterminal `$nt` as their right-hand-side; in other words, all ways to make `$nt`.

⟨Go through the raw lines of BNF grammar, stringing together, and spotting nonterminals 2⟩ ≡

```
    my $i;
    my $l;
    my $nonterminal_being_defined;
    $nonterminal_being_defined = "";
    for ($i=0; $i<$bnf_grammar_lines; $i++) {
        $l = $bnf_grammar[$i];
        if ($l =~ m/^\s*\<(.*?)\>\s*(.*)$/) {
            $nonterminal_being_defined = $1; $l = $2;
            if (not(exists $productions{$nonterminal_being_defined}))
                ⟨Create new nonterminal 3⟩;
        }
        if ($l =~ m/^\s*$/) { next; }                                    skip blank line
        if ($l =~ m/^\s*\:\=\s+(.*)$/) ⟨A new production begins here 4⟩;
        if ($l =~ m/^\s*\.\.\.\s+(.*)$/) ⟨The previous production continues onto this line 5⟩;
    }
```

This code is used in §1.

§**3.** When we see a nonterminal with a name we haven't seen before, we make it a key in the `%productions` hash, and also set its TEX representation, including a footnote identifying where in the web it originates.

⟨Create new nonterminal 3⟩ ≡
```
$productions{$nonterminal_being_defined} = $i;
$production_tex{$nonterminal_being_defined}
    = "\\nonterminal{" . $nonterminal_being_defined . "}\$_{\\rm ".
        $section_sigil[$line_sec[$bnf_grammar_source[$i]]] . "}\$";
```
This code is used in §2.

§**4.** `$productions{$nt}` is a single string, but it holds a concatenated list of productions for `$nt`. We can divide this up into its entries again because each one ends in a sentinel ampersand, `&`. (In `inweb` BNF grammar, a literal ampersand would be written out: `AMPERSAND`. So this won't cause ambiguities.)

⟨A new production begins here 4⟩ ≡
```
my $rhs = $1;
if ($nonterminal_being_defined eq "") {
    inweb_error("BNF grammar for no obvious nonterminal\n", $bnf_grammar[$i]);
}
$no_productions_for{$nonterminal_being_defined}++;
$productions_for{$nonterminal_being_defined} .= $rhs."&";
next;
```
This code is used in §2.

§**5.** A grammar line beginning with an ellipsis marks (a) a continuation of the previous line, and also (b) a place to weave a line-break in the final TEX output – which is why we combine such a line with its predecessor, but leave the ellipsis token in.

⟨The previous production continues onto this line 5⟩ ≡
```
my $more_rhs = $1;
if ($nonterminal_being_defined eq "") {
    inweb_error("BNF grammar for no obvious nonterminal\n", $bnf_grammar[$i]);
}
my $existing_rhs = $productions_for{$nonterminal_being_defined};
$existing_rhs =~ s/\&$//;
$productions_for{$nonterminal_being_defined} = $existing_rhs." ... ".$more_rhs."&";
```
This code is used in §2.

§**6.** The scan is now complete, so we know all of the nonterminals. We do a little judicious respacing of the productions, to make square and curly braces tokens in their own rights (a convenience for later), and we strip out white space. We also initialise three more hashes:

(a) `$bnf_dependencies{$nt}` is a list of the other non-terminals which occur in productions leading to `$nt` – in effect, the things whose definitions we need to know in order to understand the definition of `$nt`.
(b) `$bnf_depth{$nt}` is used in weaving to print grammar in a hierarchical way. It will eventually be a positive integer, 1, 2, 3, ..., but will be 0 for nonterminals whose depth hasn't been determined. The idea is that if X depends on Y, and X is important enough to be worth giving a hierarchical display, then Y will have depth higher than X's.
(c) `$bnf_follows{$nt}` is either blank, or else the name of a nonterminal to which `$nt` "belongs" – meaning that it will be printed in the hierarchy underneath this owner. To a human reader, `$nt` follows this owner, but is indented further rightwards.

⟨Initialise the nonterminals discovered 6⟩ ≡

```
my $nt;
foreach $nt (sort keys %productions) {
    $bnf_dependencies{$nt} = "";
    $bnf_depth{$nt} = 0;
    $bnf_follows{$nt} = "";
    $productions_for{$nt} =~ s/\[/ \[ /g;
    $productions_for{$nt} =~ s/\]/ \] /g;
    $productions_for{$nt} =~ s/\{/ \{ /g;
    $productions_for{$nt} =~ s/\}/ \} /g;
    $productions_for{$nt} =~ s/\s+/ /g;
}
```

This code is used in §1.

§**7.** This calculates the `$bnf_dependencies{$nt}` hash (see above).

⟨Work out which nonterminals depend on which other ones 7⟩ ≡

```
my $nt;
foreach $nt (sort keys %productions) {
    my $lines = $productions_for{$nt};
    while ($lines =~ m/^(.*?)\&(.*)$/) {
        my $line = $1; $lines = $2;
        while ($line =~ m/^\s*(\S+)(.*?)$/) {
            my $tok = $1;
            $line = $2;
            if ($tok =~ m/^\<(.*)\>$/) {
                $tok = $1;
                if (exists $production_tex{$tok}) {
                    $bnf_dependencies{$tok} .= $nt . ",";
                }
            }
        }
    }
}
```

This code is used in §1.

§**8.** Some nonterminals are worth making a fuss over, and printing with little hierarchical grammars of their own. (This helps us to give structure to what would otherwise be a shapeless and difficult to read mass of productions.) Such nonterminals are called "fundamental" by `inweb`. The following is, pretty shamelessly, a list of the fundamental nonterminals in Inform.

⟨Elect certain nonterminals as being fundamental 8⟩ ≡

```
    foreach $nt (sort keys %productions) {
        if (($bnf_dependencies{$nt} eq "")
            || ($nt =~ m/\-sentence$/)
            || ($nt eq "and")
            || ($nt eq "or")
            || ($nt eq "and-or")
            || ($nt eq "paragraph")
            || ($nt eq "sentence")
            || ($nt eq "condition")
            || ($nt eq "action-pattern")
            || ($nt eq "literal-value")
            || ($nt eq "type-expression")
            || ($nt eq "physical-description")
            || ($nt eq "value")) {
            $bnf_depth{$nt} = 1;
            $bnf_follows{$nt} = "";
        }
    }
```

This code is used in §1.

§**9.** So at this point fundamental nonterminals have depth 1, while all other nonterminals have depth 0. The following decides, not very efficiently, which of the un-fundamental nonterminals need to be given increased depth so as to place them below their fundamental owners in the hierarchy.

⟨Make the fundamental nonterminals the roots of trees of those depending on them 9⟩ ≡

```
    my $pass;
    for ($pass = 1; $pass <= 2; $pass++) {
        my $changed = 1;
        while ($changed == 1) {
            $changed = 0;
            my $nt;
            foreach $nt (sort keys %productions) {
                if ($bnf_depth{$nt} == 0) {
                    $scan = $bnf_dependencies{$nt};
                    $min_depth = 10000000; $unfound = 0;
                    while ($scan =~ m/^(.*?)\,(.*)$/) {
                        $scan = $2;
                        $used = $1;
                        if ($bnf_depth{$used} > 0) {
                            if ($min_depth > $bnf_depth{$used}) {
                                $min_depth = $bnf_depth{$used};
                                $min_depth_follows = $used;
                            }
                        } else { if ($pass == 1) { $unfound = 1; } }
                    }
                    if (($unfound == 0) && ($min_depth < 100000)) {
                        $bnf_depth{$nt} = $min_depth+1;
```

```
                        $bnf_follows{$nt} = $min_depth_follows;
                        $changed = 1;
                    }
                }
            }
        }
    }
```

This code is used in §1.

§**10.** Perhaps there were no fundamental nonterminals in the first place; or perhaps there were a few, but that they didn't cover the entire language. Either way, there could be nonterminals still having depth 0. We'll promote them to depth 1, so that they have a sensible margin setting when we come to weave.

⟨Any nonterminals remaining are placed at the top level 10⟩ ≡

```
    my $nt;
    foreach $nt (sort keys %productions) {
        if ($bnf_depth{$nt} == 0) {
            $bnf_depth{$nt} = 1;
        }
    }
```

This code is used in §1.

§**11. Sequencing the output.** A large grammar with many nonterminals can be very difficult to read if listed in a jumbled-up order, so the following provides the ability to control how the thing comes out: we can output certain nonterminals of our choice first, and then ultimately follow up with the rest.

```
sub weave_bnf_grammar {
    my $toshow = $_[0];
    if ($bnf_already_parsed == 0) { parse_bnf_grammar(); $bnf_already_parsed = 1; }
    ⟨Handle a comma-separated list of commands 12⟩;
    ⟨Handle a command to show nonterminals including given text 13⟩;
    ⟨Handle a command to show a specific named nonterminal 14⟩;
    ⟨Handle a command to show all remaining nonterminals 15⟩;
    ⟨Finish up and check for errors 16⟩;
}
```

§**12.** A comma-separated list of commands is acted on from left to right:

⟨Handle a comma-separated list of commands 12⟩ ≡

```
    if ($toshow =~ m/^\: (.*?)\,\s*(.*)$/) {
        my $a = $1;
        my $b = $2;
        weave_bnf_grammar(": ".$a);
        weave_bnf_grammar(": ".$b);
        return;
    }
```

This code is used in §11.

§13. A dash indicates a partial match with the production name is enough. (For instance, to pick up every nonterminal with a name ending "-constant".)

⟨Handle a command to show nonterminals including given text 13⟩ ≡
```
if ($toshow =~ m/^\:\s*\-(.*)$/) {
    $tomatch = $1;
    foreach $nt (sort keys %productions) {
        if ($nt =~ m/\-$tomatch$/) {
            weave_nonterminal($nt);
        }
    }
    return;
}
```

This code is used in §11.

§14.

⟨Handle a command to show a specific named nonterminal 14⟩ ≡
```
if ($toshow =~ m/^\:\s*(.*)$/) {
    weave_nonterminal($1);
    return;
}
```

This code is used in §11.

§15.

⟨Handle a command to show all remaining nonterminals 15⟩ ≡
```
weave_nonterminal("value");
my $nt;
foreach $nt (sort keys %productions) {
    if ($bnf_depth{$nt} == 1) {
        weave_nonterminal($nt);
    }
}
```

This code is used in §11.

§16.

⟨Finish up and check for errors 16⟩ ≡
```
foreach $nt (sort keys %productions) {
    if ($production_done{$nt} == 0) {
        print "Error in weave of BNF grammar: omitted ", $production_tex{$nt}, "\n";
        print $nt, ": ", $bnf_depth{$nt}, " ", $bnf_follows{$nt}, " ",
            $bnf_dependencies{$nt}, "\n";
    }
}
foreach $nt (sort keys %unknown_productions) {
    print WEAVEOUT "\\medskip{\\bf Unknown production:} \\nonterminal{", $nt, "}\\par\n";
}
```

This code is used in §11.

§**17. Weaving a single production.** In other words, writing out the necessary TEX to display the grammar matching a single concept. Following each production are the ones used first in the course of defining it, and so on.

```
sub weave_nonterminal {
    my $nt = $_[0];
    if ($production_done{$nt} == 1) { return; }
    if ($bnf_depth{$nt} == 1) {
        print WEAVEOUT "\\smallbreak\\hrule\\smallbreak\n";
    } else {
        print WEAVEOUT "\\smallbreak\n";
    }
    ⟨Weave the actual nonterminal 19⟩;
    ⟨Weave grammars for any nonterminals in the hierarchy below this one 18⟩;
    $production_done{$nt} = 1;
}
```

§**18.**

⟨Weave grammars for any nonterminals in the hierarchy below this one 18⟩ ≡

```
    my $dep;
    foreach $dep (sort keys %productions) {
        if ($bnf_follows{$dep} eq $nt) {
            weave_nonterminal($dep);
        }
    }
```

This code is used in §17.

§**19.** There are various hacks here which are entirely to make the Inform grammar look prettier; I do not much repent.

⟨Weave the actual nonterminal 19⟩ ≡

```
    bnf_indent($bnf_depth{$nt} - 1);
    print WEAVEOUT $production_tex{$nt}, "\n";
    $lines = $productions_for{$nt};
    $joined_lines = 0;
    if ($nt eq "debugging-aspect") { $joined_lines = 1; $join_count = 3; }
    if ($nt eq "determiner") { $joined_lines = 1; $join_count = 2; }
    $lc = 0; $lcm = $join_count - 1;
    while ($lines =~ m/^(.*?)\&(.*)$/) {
        $lc++; $line = $1; $lines = $2;
        ⟨Get to the correct margin position for the production 20⟩;
        ⟨Weave out the tokens of the production 21⟩;
        print WEAVEOUT "\n";
    }
```

This code is used in §17.

§**20.**

⟨Get to the correct margin position for the production 20⟩ ≡

```
if ($joined_lines == 1) {
    $lcm++; if ($lcm == $join_count) { $lcm = 0; }
    if ($lc == 1) {
        print WEAVEOUT "\\par";
        bnf_indent($bnf_depth{$nt} - 1);
        print WEAVEOUT "\\qquad ::\$=\$ ";
    } else {
        if ($lcm == 0) {
            print WEAVEOUT "\\par";
            bnf_indent($bnf_depth{$nt} - 1);
            print WEAVEOUT "\\qquad \\phantom{::\$=\$} \\| ";
        } else {
            print WEAVEOUT "\\| ";
        }
    }
} else {
    print WEAVEOUT "\\par";
    bnf_indent($bnf_depth{$nt} - 1);
    print WEAVEOUT "\\qquad ::\$=\$ ";
}
```

This code is used in §19.

§**21.** The term "token" is used loosely here, and not in its technical context free grammar sense. A production at this point is a sequence of textual globs divided by a single space, such as this:

```
COMMA while <condition> [ SEMICOLON ]
```

The tokens are simply these globs, so here there are six tokens. `<condition>` here is a nonterminal.

⟨Weave out the tokens of the production 21⟩ ≡

```
$last_tok = "";
while ($line =~ m/^\s*(\S+)(.*?)$/) {
    $tok = $1;
    $line = $2;
    ⟨Insert inter-token space where appropriate 22⟩;
    $last_tok = $tok;
    ⟨Weave an ellipsis token 23⟩;
    ⟨Weave a nonterminal token 24⟩;
    ⟨Weave a literal text token 25⟩;
}
```

This code is used in §19.

§**22.**

⟨Insert inter-token space where appropriate 22⟩ ≡

```
if (($last_tok ne "[") && ($last_tok ne "{") && ($last_tok ne "")
    && ($tok ne "]") && ($tok ne "}")) {
    print WEAVEOUT " ";
}
```

This code is used in §21.

§**23.** An ellipsis is a continuation onto the next line:

⟨Weave an ellipsis token 23⟩ ≡
```
if ($tok eq "...") {
    print WEAVEOUT "\\par";
    bnf_indent($bnf_depth{$nt} - 1);
    print WEAVEOUT "\\qquad \\phantom{::\$=\$} ";
    next;
}
```

This code is used in §21.

§**24.**

⟨Weave a nonterminal token 24⟩ ≡
```
if ($tok =~ m/^\<(.*)\>$/) {
    $tok = $1;
    if (exists $production_tex{$tok}) {
        print WEAVEOUT $production_tex{$tok};
    } else {
        if ($tok =~ m/^(.*)-name$/) {
            print WEAVEOUT "{\\rm " . $1 . "}";
        } else {
            if ($tok =~ m/^(.*)-usage$/) {
                print WEAVEOUT "{\\it " . $tok . "}";
            } else {
                print WEAVEOUT "\\nonterminal{\\rm !!!!" . $tok . "!!!!}";
                $unknown_productions{$tok}++;
            }
        }
    }
    next;
}
```

This code is used in §21.

§**25.** And otherwise we have literal text in boldface, though the fact that curly braces have to be rendered in math mode complicates things a little.

⟨Weave a literal text token 25⟩ ≡
```
if ($tok eq "{") { $tok = "\$\\{\$"; }
if ($tok eq "}") { $tok = "\$\\}\$"; }
if ((length($tok) > 2) &&
    ($tok =~ m/^[A-Z\-][A-Z\-0-9]+$/)) {
    print WEAVEOUT "{\\eightbf ", $tok, "}";
    $lexical_productions{$tok}++;
} else {
    print WEAVEOUT "{\\bf ", $tok, "}";
}
```

This code is used in §21.

§**26.** Indentation, the crude way:

```
sub bnf_indent {
    my $dep = $_[0];
    my $d;
    for ($d=1; $d<=$dep; $d++) {
        print WEAVEOUT "\\quad";
    }
}
```

The Tangler 3/tang

Purpose

To write a portion of the code in a compilable form.

§**1. The Master Tangler.** Here's what has happened so far, on a `-tangle` run of `inweb`: on any other sort of run, of course, we would never be in this section of code. The web was read completely into memory, and then fully parsed, with all of the arrays and hashes populated. Program Control then sent us straight here for the tangling to begin:

```
sub tangle_source {
    my $target = $_[0];
    my $dest_file = $_[1];
    my $i;
    if ($target > 0) { print "Tangling independent target $target to $dest_file\n"; }
    language_set($tangle_target_language[$target]);
    if (language_tangles() == 0) {                                  for documentation webs, for instance
        inweb_fatal_error("can't tangle material in the language '".
            $bibliographic_data{"Language"}."'");
    }
    open(TANGLEOUT, ">".$dest_file) or die "inweb: can't open tangle file '$dest_file' for output";
    print TANGLEOUT language_shebang();
    if ($web_language != $I7_LANGUAGE) {
        print TANGLEOUT language_comment("Tangled output generated by inweb: do not edit");
    }
    for ($i=0; $i<$no_lines; $i++) {
        $line_suppress_tangling[$i] = 0;
        if ($line_category[$i] == $COMMAND_LCAT) {
            $line_suppress_tangling[$i] = 1;
        }
        if ($section_tangle_target[$line_sec[$i]] != $target) {
            $line_suppress_tangling[$i] = 1;
        }
    }
    if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE))
        ⟨Pull forward the inclusion of ANSI C libraries 2⟩;
    ⟨Tangle all the constant definitions in section order 3⟩;
    if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE))
        ⟨Tangle the structure definitions 4⟩;
    ⟨Tangle the paragraphs appearing above the bar in each section in turn 6⟩;
    if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE))
        ⟨Tangle predeclarations of C functions 7⟩;
    tangle_code(0, $no_lines-1, 0);
    close(TANGLEOUT);
}
```

§**2.** For C only: we need to include ANSI library files before declaring structures because otherwise `FILE` and suchlike types won't exist yet. It might seem reasonable to include all of the `#include` files right now, but that defeats any conditional compilation, which Inform (for instance) needs in order to make platform-specific details to handle directories without POSIX in Windows. So we'll just advance the common ANSI inclusions.

⟨Pull forward the inclusion of ANSI C libraries 2⟩ ≡

```
my $i;
for ($i=0; $i<$no_lines; $i++) {
    if ($line_text_raw[$i] =~ m/^\s*\#include\s+\<(.*?)\.h\>\s*$/) {
        $clib = $1;
        if (($clib eq "stdio") ||
            ($clib eq "ctype") ||
            ($clib eq "math") ||
            ($clib eq "stdarg") ||
            ($clib eq "stdlib") ||
            ($clib eq "string") ||
            ($clib eq "time")) {
            $line_now_tangled[$i] = 1;
            print TANGLEOUT $line_text_raw[$i], "\n";
        }
    }
}
```

This code is used in §1.

§**3.** This is the result of all those `@d` definitions, and the tricky part is that tangling them depends on how constants are defined in the current programming language; we also want to expand `[[Author]]`-like substitutions in them.

⟨Tangle all the constant definitions in section order 3⟩ ≡

```
my $i;
for ($i=0; $i<$no_lines; $i++) {
    if (($line_category[$i] == $BEGIN_DEFINITION_LCAT) &&
        ($target == $section_tangle_target[$line_sec[$i]])) {
        my $definition_fragment = $line_operand_2[$i];
        $line_suppress_tangling[$i] = 1;
        language_start_definition($line_operand[$i],
            expand_double_squares($definition_fragment, 0));
        $i++;
        while (($i<$no_lines) && ($line_category[$i] == $CONT_DEFINITION_LCAT)) {
            language_prolong_definition(expand_double_squares($line_text_raw[$i], 0));
            $line_suppress_tangling[$i] = 1;
            $i++;
        }
        language_end_definition();
        $i--;
    }
}
```

This code is used in §1.

§**4.** Structures must be declared in order such that if A incorporates at least one copy of B then B must be declared before A (since C compilers require this). To track this, we have already formed structures into a directed acyclic graph (DAG): a conceptual picture in which each structure forms a "vertex" and each incorporation of B within A makes an "edge" from A to B. (If it contains multiple copies of B, there will be multiple such edges, so properly speaking this is a multigraph.) As the following runs,

(a) The current vertices are those structure names `S` for which `$structure_declaration_tangled{S}` is still 0.
(b) The current edges outward from `S` are stored in `$structure_incorporates{S}` in the form of a list of destination vertices `R`, `T`, ..., as a string `->R->T...`. Each of these must be a current vertex.
(c) At any given time, either the DAG is empty or there is at least one vertex with no outward edges: for if not, then start from a vertex v_1; it must have an outward edge to $v_2 \neq v_1$ (for otherwise there a structure contains itself, which is impossible); v_2 must also have an outward edge to v_3, and so on. If it is never the case that $v_i = v_j$ for any $i \neq j$ then there are an infinite number of structures, which is impossible. So (suppose $i < j$) there exists a loop $v_i, v_{i+1}, v_{i+2}, ..., v_j, v_i$ showing that structure i incorporates an embedded copy of itself, which is again impossible.

The proof (c) gives us a fairly efficient way to find a vertex with no outward edges, but we won't do it that way: the number of vertices is small and the number of edges always likely to be small compared to that, so it is more efficient to sweep through the DAG repeatedly removing all vertices without outward edges.

⟨Tangle the structure definitions 4⟩ ≡

```
my $no_structs_left = 0;
foreach $struc (keys %structures) {
    my $j;
    $no_structs_left++;
    $structure_declaration_tangled{$struc} = 0;
}
while ($no_structs_left > 0) {
    my $changes_made = 0;
    foreach $struc (sort keys %structures) {
        if (($structure_incorporates{$struc} eq "") &&                    no outward edges
            ($structure_declaration_tangled{$struc} == 0))                but still in graph
            ⟨The structure is a vertex with no outward edges, so declare it now 5⟩;
    }
    if ($changes_made == 0) {
        inweb_error("cyclic error in structure dependencies");
        foreach $struc (sort keys %structures) {
            if ($structure_declaration_tangled{$struc} == 0) {
                inweb_error($struc." needs prior declaration of ".
                    $structure_incorporates{$struc});
            }
        }
        exit(1);
    }
}
```

This code is used in §1.

§**5.** Here we have a vertex in our DAG which has no outward edges, so we can make its declaration and then (i) delete all inward edges and (ii) delete the vertex itself, thus leaving the DAG strictly smaller and still a well-formed DAG.

⟨The structure is a vertex with no outward edges, so declare it now 5⟩ ≡

Tangle the `typedef` *declaration lines...*

```
my $j;
for ($j=$structure_declaration_line{$struc}; $j<=$structure_declaration_end{$struc}; $j++) {
    print TANGLEOUT $line_text_raw[$j], "\n";
    $line_suppress_tangling[$j] = 1;
}
```

...and keep the counts accurate:

```
$no_structs_left--; $changes_made++;
```

(i) Strike this structure name out of the dependency lists of all others...

```
foreach $n2 (sort keys %structures) {
    $structure_incorporates{$n2} =~ s/\-\>$struc\b//g;
}
```

(ii) Remove this structure from further consideration

```
$structure_declaration_tangled{$struc} = 1;
```

This code is used in §4.

§**6.** We have no need to loop through the sections; the lines already appear in the line list in section order. We let through everything between a `@Definitions:` heading and the bar into the tangler:

⟨Tangle the paragraphs appearing above the bar in each section in turn 6⟩ ≡

```
my $i;
for ($i=0; $i<$no_lines; $i++) {
    if ($line_category[$i] == $DEFINITIONS_LCAT) {
        $i++;
        my $md_from = $i;
        while (($i<$no_lines) && ($line_category[$i] != $BAR_LCAT)) { $i++; }
        my $md_to = $i;
        tangle_code($md_from, $md_to, 0);
        for ($i=$md_from; $i<$md_to; $i++) { $line_suppress_tangling[$i] = 1; }
        $i--;
    }
}
```

This code is used in §1.

§**7.** In a rather unnecessary way, we choose to predeclare the functions in a human-readable order, for the benefit of anyone checking the tangled source: so they are neatly arranged by chapter and section. It does mean that the running time of the following is $O(C \times S \times F)$, where C, S and F are the number of chapters, sections and functions respectively, and this is in principle cubic in the size of the web. In practice the coefficient is small, of course, and even on the largest webs yet constructed there's no appreciable delay.

⟨Tangle predeclarations of C functions 7⟩ ≡

```
my $chn;
for ($chn=0; $chn<$no_chapters; $chn++) {
    print TANGLEOUT "\n",
        language_comment("Functions in chapter ".$chapter_sigil[$chn]), "\n";
    my $sn;
    for ($sn=0; $sn<$no_sections; $sn++) {
        if ($section_chap[$sn] != $chn) { next; }
        print TANGLEOUT "\n", language_comment("Section ".$section_sigil[$sn]);
        foreach $fname (sort keys %functions_line) {
            $j = $functions_line{$fname};
            if ($line_sec[$j] != $sn) { next; }
            $spc = $functions_arglist{$fname};
            $spc =~ s/\,\s*/\,\n\t\t/g;
            print TANGLEOUT $functions_return_type{$fname}, " ", $fname, "(", $spc, ");\n";
            if (exists $dot_i6_identifiers{$fname}) {
                print TANGLEOUT language_comment("accessed via .i6 metalanguage");
            } else {
                print TANGLEOUT $functions_usage_verbosely_described{$fname};
            }
        }
    }
}
```

This code is used in §1.

§**8. The real work.** All of the above was merely a series of teasing preliminaries: here's the real thing, the single recursive routine which tangles the code.

```
sub tangle_code {
    my $from = $_[0];                                   Start line
    my $to = $_[1];                                     End line
    my $permit_macro = $_[2];                           Allow CWEB macro definition bodies through?
    my $tline;
    my $contiguous = 0;
    for ($tline=$from; $tline<=$to; $tline++) {
        if (($permit_macro == 0) && ($line_occurs_in_CWEB_macro_definition[$tline] == 1)) {
            $contiguous = 0; next;
        }
        if ($line_suppress_tangling[$tline] == 1) { $contiguous = 0; next; }
        if (not(($line_category[$tline] == $CODE_BODY_LCAT))) { $contiguous = 0; next; }
        if ($line_text_raw[$tline] =~ m/^\s*\@c/) { $contiguous = 0; next; }
        if ($line_text_raw[$tline] =~ m/^(.*?)\@\<(.*?)\@\>\s*(.*?)$/)
            ⟨Expand the CWEB macro used on this line 10⟩;
        ⟨Place a comment indicating original source file position if necessary 9⟩;
        $expanded = expand_double_squares($line_text_raw[$tline], 1);
        if ($expanded =~ m/^\?\:\s*(.*)$/) {
            inweb_error_at_program_line("C-for-Inform error ($1)", $tline);
        }
        tangle_out($expanded);
    }
}
```

§**9.** If order for C compilers to report C syntax errors on the correct line, despite rearranging by automatic tools, C conventionally recognises the preprocessor directive `#line` to tell it that a contiguous extract follows from the given file; we generate this automatically. (In its usual zany way, Perl recognises the exact same syntax, thus in principle overloading its comment notation `#`.)

⟨Place a comment indicating original source file position if necessary 9⟩ ≡

```
    if (($contiguous == 0) &&
        (($web_language == $C_LANGUAGE) ||
            ($web_language == $C_FOR_INFORM_LANGUAGE) ||
            ($web_language == $PERL_LANGUAGE))) {
        $contiguous = 1;
        tangle_out("#line ".$line_source_file_line[$tline].
            " \"".$section_pathname_relative_to_web[$line_sec[$tline]]."\"");
    }
```

This code is used in §8,10,8.

§**10.** The tricky point here is that CWEB is stream-of-characters rather than line-oriented, which makes it awkward declaring to the C preprocessor exactly where the material came from without inserting a lot of line breaks: because the C preprocessor believes each line in the final C code must have a single file and line as its point of origin, and that just isn't true if the original CWEB code read, say,

```
if (black == white) { @<Observe the universe's predicament@>; exit(1); }
```

Happily C is pretty relaxed about treating newlines as just more white space, so the rather spurious newlines cause no trouble.

Because we work on lines, not a stream, it isn't very convenient for us to make more than one macro substitution on the same line. We could fix this if we wanted to, but in practice it just doesn't naturally arise, because they tend to have rather long descriptive names.

A point of difference with CWEB is that we automatically encase the expanded macro in braces, for C or Perl, at any rate. This is convenient since it allows us to treat a CWEB macro as a single statement; and also usefully prevents us from some of the abusive things done in *TEX: The Program*, where Knuth uses CWEB macros to combine bits of top-level global variable definitions and other such outside-of-functions hackery (to be fair, Knuth was forced into this by the deficiencies of Pascal as it then stood).

⟨Expand the CWEB macro used on this line 10⟩ ≡

```
my $prologue = $1; local $name = $2; local $residue = $3;
if ($residue =~ m/^(.*?)\@\<(.*?)\@\>(.*)$/) {
    inweb_error_at_program_line("only one <...> macro can be used on any single line",
        $tline);
}
⟨Place a comment indicating original source file position if necessary 9⟩;
tangle_out(expand_double_squares($prologue, 1));
if (not(exists $cweb_macros_start{$name})) {
    inweb_error_at_program_line("no such <...> macro as '".$name."'", $tline);
} else {
    if (($web_language == $C_LANGUAGE) ||
        ($web_language == $C_FOR_INFORM_LANGUAGE) ||
        ($web_language == $PERL_LANGUAGE)) { print TANGLEOUT " { "; }
    tangle_out(expand_double_squares($cweb_macros_surplus_bit{$name}, 1));
    tangle_code($cweb_macros_start{$name}, $cweb_macros_end{$name}, 1);
    if (($web_language == $C_LANGUAGE) ||
        ($web_language == $C_FOR_INFORM_LANGUAGE) ||
        ($web_language == $PERL_LANGUAGE)) { print TANGLEOUT " } "; }
}
$contiguous = 0;
print TANGLEOUT "\n";
if ($residue ne "") {
    ⟨Place a comment indicating original source file position if necessary 9⟩;
    tangle_out(expand_double_squares($residue, 1));
}
next;
```

This code is used in §8.

§**11.**

```
sub tangle_out {
    my $line_of_code = $_[0];
    if ($web_language == $C_FOR_INFORM_LANGUAGE) {
        $line_of_code =~ s/\:\:/__/g;
    }
    print TANGLEOUT $line_of_code, "\n";
}
```

§**12. Substituting bibliographic data.** We expand material in double square brackets if either

(a) we recognise it as the name of a piece of bibliographic data, or
(b) it is one of the C-for-Inform language extensions and in a code context where this would make sense.

Otherwise, we leave well alone.

```
sub expand_double_squares {
    my $line = $_[0];
    my $with_extensions = $_[1];
    my $fore;
    my $aft;
    my $comm;
    my $expanded;
    my $safety_check = 0;
    my $so_far = "";
    while ($line =~ m/^(.+?)\[\[(.*?)\]\](.*)$/) {
        $fore = $1; $line = $3; $comm = $2;
        $so_far .= $fore;
        if (exists ($bibliographic_data{$comm})) {
            $so_far .= $bibliographic_data{$comm};
        } elsif (($with_extensions == 1) && ($web_language == $C_FOR_INFORM_LANGUAGE)) {
            $expanded = expand_double_squared_command($comm);
            if ($expanded =~ m/^\?\: /) { return $expanded; }
            $so_far .= $expanded;
            if ($safety_check++ == 100) {
                return '?: expander gone into infinite regress';
            }
        } else {
            $so_far .= '[['.$comm.']]';
        }
    }
    $so_far .= $line;
    return $so_far;
}
```

Programming Languages 3/plan

Purpose

To characterise the relevant differences in behaviour between the various programming languages we support: for instance, how comments are represented.

Definitions

¶1. At any given time, there's a current programming language, and it is always one of the following possibilities:

```
define $C_LANGUAGE 1                          C or C++
define $C_FOR_INFORM_LANGUAGE 2               Ditto, but with NI extensions
define $PERL_LANGUAGE 3                       Perl
define $I6_LANGUAGE 4                         Inform 6
define $I7_LANGUAGE 5                         Inform 7 extension
define $PLAIN_LANGUAGE 6                      Plain text
define $NO_LANGUAGE 100                       Nothing to tangle, by fiat

$web_language = $C_LANGUAGE;                  The default
$tangled_extension = ".c";
```

---

§**1.** Setting the current language from a textual form:

```
sub language_set {
    my $lname = $_[0];
    $web_language = -1;
    if ($lname eq "C") { $web_language = $C_LANGUAGE; $tangled_extension = ".c"; }
    if ($lname eq "C++") { $web_language = $C_LANGUAGE; $tangled_extension = ".cpp"; }
    if ($lname eq "C for Inform") {
        $web_language = $C_FOR_INFORM_LANGUAGE; $tangled_extension = ".c";
        ⟨Flag some identifiers as unusual in this form of C 2⟩;
    }
    if ($lname eq "Perl") { $web_language = $PERL_LANGUAGE;  $tangled_extension = ".pl"; }
    if ($lname eq "Inform 6") { $web_language = $I6_LANGUAGE;  $tangled_extension = ".i6"; }
    if ($lname eq "Inform 7") { $web_language = $I7_LANGUAGE;  $tangled_extension = ".i7x"; }
    if ($lname eq "Plain Text") { $web_language = $PLAIN_LANGUAGE;  $tangled_extension = ".txt"; }
    if ($lname eq "None") { $web_language = $NO_LANGUAGE;  $tangled_extension = ""; }
    if ($web_language == -1) {
        inweb_fatal_error("unsupported programming language ".$lname);
    }
}
```

§**2.** The term "blacklisting" is a little melodramatic. A function which has been blacklisted is allowed to be defined more than once in the source code; Inform needs this for `isdigit`, of all things, because the Windows library's implementation is deficient.

Similarly, a member name which has been blacklisted is allowed to be present in more than one structure. This of course is perfectly legal in C, but the Inform source code tries to avoid doing it, since it leads to confusion and makes it harder to nail down where the structure is used. A few exemptions are made for members deliberately used in common across wide ranges of structures.

⟨Flag some identifiers as unusual in this form of C 2⟩ ≡

```
    $blacklisted_functions{"isdigit"} = 1;
    $blacklisted_members{"word_ref1"} = 1;
    $blacklisted_members{"word_ref2"} = 1;
    $blacklisted_members{"next"} = 1;
    $blacklisted_members{"down"} = 1;
    $blacklisted_members{"allocation_id"} = 1;
    $members_allowed_to_be_unused{"handling_routine"} = 1;
```

This code is used in §1.

§**3.** The file extension generally used by files of code in this language:

```
sub language_file_extension { return $tangled_extension; }
```

§**4.** Any compulsory heading that must occur on line 1, really:

```
sub language_shebang {
    my $text = $_[0];
    if ($web_language == $PERL_LANGUAGE) { return "#!/usr/bin/perl\n\n"; }
    return "";
}
```

§**5.** Now a routine to write a comment. Languages without comment should write nothing.

```
sub language_comment {
    my $text = $_[0];
    if ($web_language == $C_LANGUAGE) { return "/* ".$text." */\n"; }
    if ($web_language == $C_FOR_INFORM_LANGUAGE) { return "/* ".$text." */\n"; }
    if ($web_language == $PERL_LANGUAGE) { return "# ".$text."\n"; }
    if ($web_language == $I6_LANGUAGE) { return "! ".$text."\n"; }
    if ($web_language == $I7_LANGUAGE) { return "[".$text."]\n"; }
    return "";
}
```

§**6.** And we need to spot and so on... comments:

```
sub language_and_so_on {
    my $text = $_[0];
    if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE)) {
        if ($text =~ m/^\s*\/\* ...and so on... \*\/\s*/) { return 1; }
    }
    if ($web_language == $I6_LANGUAGE) {
        if ($text =~ m/^\s*\!\s*...and so on...\s*$/) { return 1; }
    }
    if ($web_language == $I7_LANGUAGE) {
        if ($text =~ m/^\s*\[\s*...and so on...\s*\]\s*$/) { return 1; }
    }
    return 0;
}
```

§**7.** And in general to see when a line ends in a comment:

```
sub line_ends_with_comment {
    my $matter = $_[0];
    if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE)) {
        if ($matter =~ m/^(.*)\/\*\s*(.*?)\s*\*\/\s*$/) {
            $part_before_comment = $1; $part_within_comment = $2; return 1;
        }
    }
    if ($web_language == $PERL_LANGUAGE) {
        if ($matter =~ m/^\#\s+(.*?)\s*$/) {
            $part_before_comment = ""; $part_within_comment = $1; return 1;
        }
        if ($matter =~ m/^(.*)\s+\#\s+(.*?)\s*$/) {
            $part_before_comment = $1; $part_within_comment = $2; return 1;
        }
    }
    return 0;
}
```

§**8.** To place page breaks strategically within code:

```
sub language_pagebreak_comment {
    my $l = $_[0];
    if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE)) {
        if ($l =~ m/^\s*\/\* PAGEBREAK \*\/\s*$/) { return 1; }
    }
    if ($web_language == $PERL_LANGUAGE) {
        if ($l =~ m/^\s*\#\s+PAGEBREAK\s*$/) { return 1; }
    }
    if ($web_language == $I6_LANGUAGE) {
        if ($l =~ m/^\s*\!\s+PAGEBREAK\s*$/) { return 1; }
    }
    if ($web_language == $I7_LANGUAGE) {
        if ($l =~ m/^\s*\[\s*PAGEBREAK\s*\]\s*$/) { return 1; }
    }
    return 0;
}
```

§**9.** Does the currently selected language allow tangling? (Yes, unless this is purely a documentation project.)

```
sub language_tangles {
    if ($web_language == $NO_LANGUAGE) { return 0; }
    return 1;
}
```

§**10.** The following routines handle the `@d` escape, writing a definition of the constant named `$term` as the value given. If the value spans multiple lines, the first-line part is supplied to `language_start_definition` and then subsequent lines are fed in order to `language_prolong_definition`. At the end, `language_end_definition` is called.

```
sub language_start_definition {
    my $term = $_[0];
    my $startval = $_[1];
    if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE)) {
        if ($web_language == $C_FOR_INFORM_LANGUAGE) { $startval =~ s/::/__/g; }
        print TANGLEOUT "#define ", $term, " ", $startval;
        return;
    }
    if ($web_language == $PERL_LANGUAGE) {
        print TANGLEOUT $term, " = ", $startval;
        return;
    }
    inweb_error("programming language $web_language does not support \@d");
}
sub language_prolong_definition {
    my $more = $_[0];
    if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE)) {
        if ($web_language == $C_FOR_INFORM_LANGUAGE) { $more =~ s/::/__/g; }
        print TANGLEOUT "\\\n    ", $more;
        return;
```

```
    }
    inweb_error("programming language $web_language does not support multiline \@d");
}
sub language_end_definition {
    if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE)) {
        print TANGLEOUT "\n";
    }
    if ($web_language == $PERL_LANGUAGE) {
        print TANGLEOUT "\n;\n";
    }
}
```

Purpose

To provide a convenient extension to C syntax for the C-for-Inform language, which is likely never to be used for any program other than the Inform 7 compiler.

§**1. Unit testing.** The following provides a simple test of the extended syntax, and is run by using the `-test-extensions` switch at the command line. It gives some examples of what we'll be compiling below, anyway.

```
sub full_test_double_squares {
    test_ds("[[w1, w2 == ...]]");
    test_ds("[[w1, w2 == golly]]");
    test_ds("[[word w1 == zowie]]");
    test_ds("[[word w1 == wow/huh/zowie]]");
    test_ds("[[w1, w2 == by the pool]]");
    test_ds("[[w1, w2 == moreover ***]]");
    test_ds("[[w1, w2 == *** by the pool]]");
    test_ds("[[w1, w2 == by the pool ***]]");
    test_ds("[[w1, w2 == golly gosh/moses mrs smith]]");
    test_ds("[[w1, w2 == so mr ### we meet again]]");
    test_ds("[[w1, w2 == hello there ...]]");
    test_ds("[[w1, w2 == ... the memory of water]]");
    test_ds("[[w1, w2 <-- something]]");
    test_ds("[[w1, w2 == ... nantucket ...]]");
    test_ds("[[w1, w2 == ... when ... : w]]");
    test_ds("[[w1, w2 == hot porridge ... when ... : w]]");
    test_ds("[[w1, w2 == ... when ... breakfast is served]]");
    test_ds("[[w1, w2 == ... when ... and ... breakfast is served]]");
    test_ds("[[w1, w2 == ... when ... and ... breakfast is served : w]]");
    test_ds("[[w1, w2 == ... when ... and ... breakfast is served : w, x]]");
    test_ds("[[w1, w2 == ... when ... and ... breakfast is served : w, x, y]]");
    test_ds("[[w1, w2 == ... when ... breakfast is served : w, z, x]]");
    test_ds("[[w1, w2 == bacon ... when ... breakfast is served : w]]");
    test_ds("[[w1, w2 == and now ... --> now1, now2]]");
    test_ds("[[w1, w2 == ... until then --> then1, then2]]");
    test_ds("[[w1, w2 == ... when ... : w --> w1, w2 ... when1, when2]]");
    test_ds("[[w1, w2 == ... when ... : w --> when1, when2 ... w1, w2]]");
    test_ds("[[w1, w2 == ... when ... : w --> when1, w2 ... when2, w1]]");
    test_ds("[[w1, w2 == ... OPENBRACKET for ... only CLOSEBRACKET --> x1, x2 ... vm1, vm2]]");
}

sub test_ds {
    my $original = $_[0];
    print "inweb: testing expansion ", $original, "\n\n";
    print expand_double_squares(" ".$original), "\n\n";
}
```

§**2. The expander.** We are given what might, or might not, be a valid command. If we can make sense of it, we return its expansion; if not, we return an error message with a query ? in column 1.

```
sub expand_double_squared_command {
    my $comm = $_[0];
    my $operator = 0;
    my $lhs;
    my $rhs;
    my $w1;
    my $w2;
    my $single_word_flag = 0;
    ⟨Identify the command as either a test or an assignment 3⟩;
    ⟨Parse the left-hand side into C expressions for a word range 4⟩;
    if ($operator == 1) {
        if ($single_word_flag == 1) ⟨Expand a single word test 6⟩
        else ⟨Expand a word range test 7⟩;
    }
    if ($operator == 2) ⟨Expand an assignment command 5⟩;
    return '?: This should not happen';
}
```

§**3.** We split the line as RHS, operator, LHS; there are only two valid operators.

⟨Identify the command as either a test or an assignment 3⟩ ≡

```
    if ($comm =~ m/^(.*?)\s*\=\=\s*(.*?)\s*$/) {
        $operator = 1; $lhs = $1; $rhs = $2;
    } else {
        if ($comm =~ m/^(.*?)\s*\<\-\-\s*(.*?)\s*$/) {
            $operator = 2; $lhs = $1; $rhs = $2;
        }
    }
    if ($operator == 0) {
        return '?: [[ ... ]] with neither <-- nor == operators';
    }
```

This code is used in §2.

§**4.**

⟨Parse the left-hand side into C expressions for a word range 4⟩ ≡

```
    if ($lhs =~ m/^\s*(\S+?)\s*\,\s*(\S+)\s*$/) {
        $w1 = $1; $w2 = $2;
    } else {
        if ($lhs =~ m/^\s*(\S+)\s*$/) {
            $w1 = $1."->word_ref1"; $w2 = $1."->word_ref2";
        } else {
            if ($lhs =~ m/^\s*word\s+(\S+)\s*$/) {
                $w1 = $1; $w2 = $w1; $single_word_flag = 1;
            } else {
                return '?: [[ ... ]] without comma-separated word pair';
            }
        }
    }
```

This code is used in §2.

§**5.** There are three basic commands, of which one is easy:

⟨Expand an assignment command 5⟩ ≡
```
if ($single_word_flag == 1) {
    return '?: [[ ... ]] cannot assign a single word variable';
}
return $w1 . " = " . $rhs . "->word_ref1; ". $w2 . " = " . $rhs . "->word_ref2;";
```
This code is used in §2.

§**6.** And one is nearly easy:

⟨Expand a single word test 6⟩ ≡
```
if ($rhs =~ m/^\s*(\S+?)\s*$/) {
    my $list = $1;
    my $cond = '((' .$w1.'>=0) && (';
    my $ct = 1;
    while ($list =~ m/^([^\/]+)(.*)$/) {
        if ($ct > 1) { $cond .= ' || '; }
        $ct++;
        $cond .= compare_word_p($w1, 0, $1);
        $list = $2; $list =~ s/^\///;
    }
    $cond .= '))';
    return $cond;
}
return '?: [[ ... ]] cannot test a single word variable to other than a single word';
```
This code is used in §2.

§**7.** While the third is the big one. We are now comparing a word range against what may be a complicated pattern.

⟨Expand a word range test 7⟩ ≡
```
my $write_positions = "";
my $write_ranges = "";
⟨Split off optional portions of the pattern indication what positions or ranges to write 8⟩;
my $right_unanchored = 0;
my $left_unanchored = 0;
⟨Determine whether the pattern is anchored to the right or left edges 9⟩;
my $nw = 0;
⟨Split the pattern into an array of words 10⟩;
my $nr = 0;
⟨Divide this up into ranges with ellipses between 11⟩;
⟨Work out how these ranges are anchored 12⟩;
if ($nr == 0) { return "(FALSE)"; }
my $cond = "(";
⟨Work out a C condition to test the pattern 13⟩;
$cond .= ")";
if ($write_ranges ne "") { return '?: [[ ... ]] has too many write pairs of variables'; }
if ($write_positions ne "") { return '?: [[ ... ]] has too many : variables'; }
return $cond;
```
This code is used in §2.

§**8.**

⟨Split off optional portions of the pattern indication what positions or ranges to write 8⟩ ≡

```
if ($rhs =~ m/^(.*)\s*\-\-\>\s*(.*?)\s*$/) {
    $write_ranges = $2; $rhs = $1;
}
if ($rhs =~ m/^(.*)\s*\:\s*(.*?)\s*$/) {
    $rhs = $1; $write_positions = $2;
}
if (($rhs.' '.$write_positions) =~ m/\-\-\>/) { return '?: [[ ... ]] has too many -->s'; }
$rhs =~ m/^\s*(.*?)\s*$/; $rhs = $1;
$write_positions =~ m/^\s*(.*?)\s*$/; $write_positions = $1;
$write_ranges =~ m/^\s*(.*?)\s*$/; $write_ranges = $1;
```

This code is used in §7.

§**9.** If either end of the pattern is `***`, this means "match all words here right up to the edge"; we'll call the pattern "unanchored", because the word position where the match will start or finish is somewhere loose inside the word range we're matching.

⟨Determine whether the pattern is anchored to the right or left edges 9⟩ ≡

```
if ($rhs =~ m/^(.*)\s+\*\*\*\s*$/) {
    $rhs = $1;
    $right_unanchored = 1;
}
if ($rhs =~ m/^\*\*\*\s+(.*)$/) {
    $rhs = $1;
    $left_unanchored = 1;
}
```

This code is used in §7.

§**10.**

⟨Split the pattern into an array of words 10⟩ ≡

```
while ($rhs =~ m/^(\S*?)\s+(.*)$/) {
    $words[$nw++] = $1; $rhs = $2;
}
$words[$nw++] = $rhs;
```

This code is used in §7.

§**11.**

⟨Divide this up into ranges with ellipses between 11⟩ ≡

```
my $i;
my $range_start = -1;
for ($i=0; $i < $nw; $i++) {
    if ($words[$i] eq '...') {
        if ($range_start >= 0) {
            $range_from[$nr] = $range_start; $range_to[$nr++] = $i-1;
            $range_start = -1;
        }
    } else {
        if ($range_start == -1) { $range_start = $i; }
    }
}
if ($range_start >= 0) {
    $range_from[$nr] = $range_start; $range_to[$nr++] = $nw-1;
}
```

This code is used in §7.

§**12.**

⟨Work out how these ranges are anchored 12⟩ ≡

```
my $i;
for ($i=0; $i<$nr; $i++) {
    $range_length[$i] = $range_to[$i]-$range_from[$i]+1;
    $range_anchor[$i] = -1;
    $range_anchor_direction[$i] = 0;
    $range_from_fixed[$i] = 0;
    $range_to_fixed[$i] = 0;
    if (($range_from[$i] == 0) && ($left_unanchored == 0)) {
        $range_anchor[$i] = 0;
        $range_anchor_direction[$i] = 1;
        $range_from_fixed[$i] = 1;
    }
    if (($range_to[$i] == $nw-1) && ($right_unanchored == 0)) {
        $range_anchor[$i] = $nw-1;
        $range_anchor_direction[$i] = -1;
        $range_to_fixed[$i] = 1;
    }
}
```

This code is used in §7.

§**13.**

⟨Work out a C condition to test the pattern 13⟩ ≡

⟨Begin with a check that the word range is valid and includes enough words 14⟩;

```
my $i;
for ($i=0; $i<$nr; $i++) { $range_done[$i] = 0; }
$left_inset = 0;
$right_inset = 0;
$left_extent = 0;
$right_extent = 0;
```

⟨First check all of the ranges with fixed endpoints 15⟩;
⟨Then check all of the ranges with floating endpoints 16⟩;

```
for ($i=0; $i<$nr; $i++) {
    if ($range_done[$i] == 0) { return '?: [[ ... ]] has intractable range'; }
}
```

⟨Add always-true conditions with the side effect of writing the word ranges 17⟩;

This code is used in §7.

§**14.** Word ranges inside the Inform compiler are pairs of values (w_1, w_2) where either $w_1 = w_2 = -1$, meaning "no text", or $0 \le w_1 \le w_2 < N$, where N is the number of words read in by the lexer. Here we are testing a word range where w_1 is represented by the C expression in `$w1`, and similarly w_2 by `$w2`. So we first check that $w_1 \ge 0$, and then that not only is $w_2 \ge w_1$, but also that it's larger by sufficient to include all of the words we're trying to match. Thus

```
[[x1, x2 == unseeded ... grapes]]
```

will certainly fail if the word range contains fewer than three words.

⟨Begin with a check that the word range is valid and includes enough words 14⟩ ≡

```
$cond .= "(". $w1 . ">=0)";
my $need = 0;                                   # minimum number of words needed
my $exact = 1;                                  # is this the exact number of words we must have?
for ($i=0; $i<$nr; $i++) {
    $need = $need + $range_length[$i];
    if ($range_from_fixed[$i] == 0) { $exact = 0; }
    if ($range_to_fixed[$i] == 0) { $exact = 0; }
}
for ($i=0; $i<$nw; $i++) {
    if ($words[$i] eq '...') { $need++; }
}
$need--;
if ($exact == 1) { $op = "=="; } else { $op = ">="; }
$cond .= " && (". $w2 . $op . var_offset($w1, $need) . ")";
```

This code is used in §13.

§**15.** Suppose we have a pattern such as

```
[[x1, x2 == peeled or ... perhaps ... unpeeled mangos]]
```

This gives us three ranges to match, of lengths 2, 1, 2. We start by testing the two end ranges, because we can do so quickly – if there's going to be a match, we know exactly at what word positions they must occur. Thus, "peeled" must be at `x1`, "mangos" must be at `x2`, and so on. The reason for doing this is that it will be slower to find "perhaps", whose position is ambiguous, so we don't want to look unless the pattern otherwise matches – this enables us to reject non-matching word ranges more quickly.

⟨First check all of the ranges with fixed endpoints 15⟩ ≡

```
my $j;
for ($i=0; $i<$nr; $i++) {
    if ($range_done[$i] == 1) { next; }
    if (($range_from[$i] == 0) && ($range_from_fixed[$i] == 1)) {
        for ($j=0; $j<$range_length[$i]; $j++) {
            $cond .= compare_word($w1, $j, $words[$j]);
        }
        $range_done[$i] = 1;
        $left_extent = $range_length[$i];
        $left_inset += $range_length[$i] - 1;
        next;
    }
    if (($range_to[$i] == $nw-1) && ($range_to_fixed[$i] == 1)) {
        for ($j=$range_length[$i]-1; $j>=0; $j--) {
            $cond .= compare_word($w2, 0-$j, $words[$nw-1-$j]);
        }
        $range_done[$i] = 1;
        $right_inset += $range_length[$i] - 1;
        $right_extent = $range_length[$i];
        next;
    }
}
```

This code is used in §13.

§**16.** A floating range can consist of only a single word, at present. (An occasional nuisance when coding Inform, but basically an acceptable compromise.) We detect this with a call to Inform's routine `is_word_intermediate`, which determines whether a given word occurs at a position p such that $w_1 < p < w_2$, and returns the minimum value of p if it does, -1 if it does not. If there's only one floating range like this, we can just check the return value to see if it's non-negative. But if there are two or more floating ranges, thus:

```
[[x1, x2 == peeled and ... seeded ... sliced ... in syrup : i, j]]
```

then we need to store the position of "seeded", since we must not only check that it exists but also use that as the left end of the range of words inside which we look for "sliced". To do this, we extract the name of a C variable to store the information in, finding it in the `$write_positions` part of the pattern. Thus on a successful match, the example text would store the word position of "seeded" in `i`, and of "sliced" in `j`.

⟨Then check all of the ranges with floating endpoints 16⟩ ≡

```
my $left_var = $w1;
my $right_var = $w2;
$unwritten_var_exists = 0;
$no_written_vars = 0;
my $i;
for ($i=0; $i<$nr; $i++) {
```

```
        if ($range_done[$i] == 1) { next; }
        if ($range_length[$i] == 1) {
            $cond .= " && (";
            if ($write_positions ne "") {
                if ($write_positions =~ m/^\s*(.*?)\s*\,\s*(.*)$/) {
                    $write_var = $1; $write_positions = $2;
                } else { $write_var = $write_positions; $write_positions = ""; }
                $cond .= "(".$write_var."=";
            } else {
                if ($unwritten_var_exists == 1) {
                    return '?: [[ ... ]] has insufficient : variables';
                }
                $unwritten_var_exists = 1;
            }
            if ($write_var eq "") { $cond.= "("; }
            $cond .=
                "Text__is_word_intermediate(".$words[$range_from[$i]]."_V,"
                . var_offset($left_var, $left_inset)
                . ","
                . var_offset($right_var, 0-$right_inset).("))";
            if ($write_var ne "") {
                $cond .= ",(".$write_var.">=0))";
                $left_var = $write_var;
                $written_vars[$no_written_vars++] = $write_var;
            } else {
                $cond.= " >= 0)";
            }
            $range_done[$i] = 1;
            next;
        }
    }
```

This code is used in §13.

§**17.** That's it for testing the condition: now for the tricky part, storing the word ranges matched on a successful outcome. Note that nothing is written unless the test has succeeded. To see the difficulty here, consider

```
[[w1, w2 == ... OPENBRACKET ... CLOSEBRACKET : i --> w1, w2 ... p1, p2]]
```

which looks for a bracketed clause at the end of the word range and splits it off into (p_1, p_2). We are clearly going to want to set p_2 to w_2, but to the old value of w_2, not the new one, which is going to be just before the open-bracket. So it is important to assign these variables in the right order, so that we don't change w_2 before using it.

⟨Add always-true conditions with the side effect of writing the word ranges 17⟩ ≡

```
$no_assignments = 0;
⟨Work out which variables need to be assigned, and to what 18⟩;
for ($i=0; $i<$no_assignments; $i++) { $assignment_done[$i] = 0; }
⟨Iteratively make all possible safe assignments until all have been assigned 19⟩;
```

This code is used in §13.

§18. Let's call the variables to be written $v_1, v_2, v_3, ..., v_n$, where n will always be an even number. (In the example above, $n = 4$.)

In the following, we generate triples (v_i, v_k, x) to represent that an assignment $v_i \mapsto v_k + x$. Each triple is set up by a call to the routine `assign_set_var`.

⟨Work out which variables need to be assigned, and to what 18⟩ ≡

```
my $j = 0;
my $i;
for ($i=0; $i<$nw; $i++) {
    if ($words[$i] eq '...') {
        if ($j == 0) {
            $from_var = $w1;
            $from_offset = $left_extent;
        } else {
            $from_var = $written_vars[$j-1];
            $from_offset = 1;
        }
        if ($j<$no_written_vars) {
            $to_var = $written_vars[$j];
            $to_offset = -1;
            $j++;
        } else {
            $to_var = $w2;
            $to_offset = 0-$right_extent;
        }
        if ($write_ranges =~ m/^\s*(\S+)\s*\,\s*(\S+)\s*(.*?)\s*$/) {
            $write_from_var = $1;
            $write_to_var = $2;
            $write_ranges = $3;
            assign_set_var($write_from_var, $from_var, $from_offset);
            assign_set_var($write_to_var, $to_var, $to_offset);
            if ($write_ranges ne "") {
                if ($write_ranges =~ m/^\s*\.\.\.\s*(.*)$/) {
                    $write_ranges = $1;
                } else {
                    return '?: [[ ... ]] pairs of write vars should be divided by ...';
                }
            }
        } else {
            if ($write_ranges ne "") {
                return '?: [[ ... ]] write vars should be in pairs divided by ...';
            }
        }
    }
}
```

This code is used in §17.

§19. Now we have a mass of triples (v_i, v_k, x), each representing that an assignment $v_i \mapsto v_k + x$ must be made. But we can't make this assignment until v_i no longer appears in any unassigned triple (v_j, v_i, y), because to do so would invalidate the value of v_i used in that assignment.

We solve this iteratively, at each stage looking for the safe triples to assign. For each variable occurring in the middle position, we calculate `$necessity_count{$v}` as the number of other variables whose assignment is to `$v` plus or minus some offset. Thus, it's safe to assign to `$v` if and only if the necessity count is 0.

A case we have to be careful about is (v_i, v_i, x), where the assignment we'll make has the effect of adding x to v_i. This would be impossible ever to carry out if we regarded it as a dependency of v_i upon itself, which is why such a triple doesn't contribute to the necessity count of v_i.

⟨Iteratively make all possible safe assignments until all have been assigned 19⟩ ≡

```
my $no_done = 0;
my $safety_check = 0;                                    in case of a bug in this code, really
while ($no_done < $no_assignments) {
    ⟨Calculate the necessity count for each unassigned variable 20⟩;
    ⟨Assign all unassigned variables with necessity count 0 21⟩;
    if ($safety_check++ == 1000) ⟨Panic! Something has gone terribly wrong 22⟩;
}
```

This code is used in §17.

§20.

⟨Calculate the necessity count for each unassigned variable 20⟩ ≡

```
my $i;
for ($i=0; $i<$no_assignments; $i++) {
    $necessity_count{$var_to_read[$i]} = 0;
    $necessity_count{$var_to_assign[$i]} = 0;
}
for ($i=0; $i<$no_assignments; $i++) {
    if ($assignment_done[$i] == 0) {
        if ($var_to_read[$i] ne $var_to_assign[$i]) {
            $necessity_count{$var_to_read[$i]}++;
        }
    }
}
```

This code is used in §19.

§21.

⟨Assign all unassigned variables with necessity count 0 21⟩ ≡

```
my $i;
for ($i=0; $i<$no_assignments; $i++) {
    if (($assignment_done[$i] == 0) && ($necessity_count{$var_to_assign[$i]} == 0)) {
        $cond .= set_var($var_to_assign[$i], $var_to_read[$i], $var_to_offset[$i]);
        $assignment_done[$i] = 1;
        $no_done++;
    }
}
```

This code is used in §19.

§**22.** Well, perhaps this is overdramatising it. A simple case here might be

```
[[w1, w2 == possibly ... --> w2, w1]]
```

where we have to make the assignments $w_2 \mapsto w_1 + 1$, $w_1 \mapsto w_2$. Each variable depends on the other, and the log-jam cannot be broken without use of some additional storage (which C offers no convenient way to allocate, within a condition). No doubt with more labour we could contrive a way around this, but it's not worth it: such cases never in practice arose in the whole history of writing Inform.

⟨Panic! Something has gone terribly wrong 22⟩ ≡

```
my $i;
for ($i=0; $i<$no_assignments; $i++) {
    if ($assignment_done[$i] == 0) {
        inweb_error("  ".$var_to_assign[$i]." = ".$var_to_read[$i]." + ".$var_to_offset[$i]);
    }
}
return '?: [[ ... ]] no safe way to write these pairs of variables';
```

This code is used in §19.

§**23.** And that's it for the main routine. We needed two routines for dealing with these variable-assignment triples; first, to create one:

```
sub assign_set_var {
    my $write_var = $_[0];
    my $from_var = $_[1];
    my $from_offset = $_[2];
    if (($from_offset == 0) && ($write_var eq $from_var)) { return ""; }
    $var_to_assign[$no_assignments] = $write_var;
    $var_to_read[$no_assignments] = $from_var;
    $var_to_offset[$no_assignments] = $from_offset;
    $no_assignments++;
}
```

§**24.** And then to compile the assignment asked for:

```
sub set_var {
    my $write_var = $_[0];
    my $from_var = $_[1];
    my $from_offset = $_[2];
    if (($from_offset == 0) && ($write_var eq $from_var)) { return ""; }
    return " && (".$write_var."=".var_offset($from_var, $from_offset).")";
}
```

§**25.** These both need the following useful routine, which compiles a C expression consisting of a variable number plus a constant offset. For legibility, we compile to, say, `q1-5` rather than `q1+-5` in the case of a negative offset.

```
sub var_offset {
    my $var = $_[0];
    my $offset = $_[1];
    if ($offset == 0) { return $var; }
    if ($offset > 0) { return $var."+".$offset; }
    return $var."-".(0-$offset);
}
```

§**26.** Finally, a routine to generate the condition for testing a single word. Here we know we want to test the word whose number is stored in `$var` plus a constant `$with`; for instance, word `w2-1`, the penultimate word in the range being tested. And we want to test that this word is the one in `$with`.

The slight complication here is that a word in the form

```
apple/pear/peach
```

means "any of apple, pear or peach", so that we have to compile a compound condition.

```
sub compare_word {
    my $var = $_[0];
    my $offset = $_[1];
    my $with = $_[2];
    my $this;
    my $rest;
    my $cond;
    $cond = " && ";
    if ($with =~ m/\//) {
        $cond .= "(";
        while ($with =~ m/^(.*?)\/(.*)$/) {
            $this = $1; $with = $2;
            $cond .= compare_word_p($var, $offset, $this);
            $cond .= " || ";
        }
        $cond .= compare_word_p($var, $offset, $with);
        $cond .= ")";
    } else {
        $cond .= compare_word_p($var, $offset, $with);
    }
    return $cond;
}
```

§**27.** So it's actually *this* routine which generates the test of a single word. There's one final language feature to implement: the special word `###` matches any single word.

```
sub compare_word_p {
    my $var = $_[0];
    my $offset = $_[1];
    my $with = $_[2];
    if ($with eq '###') { return "(TRUE)"; }
    return "(compare_word(" . var_offset($var, $offset) . ", " . $with . "_V))";
}
```