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.** 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;
}
```