3 Outputs

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";
        }
    }
}
```