Solution Deviser 3/sol

Purpose

To make a solution (.sol) file accompanying a release, if requested.

3/sol.§2-13 Step 1: building the Skein tree; §14 Step 2: identify the relevant lines; §15-16 Step 3: pruning irrelevant lines out of the tree; §17-21 Step 4: writing the solution file; §22-23 Writing individual commands and branch descriptions

---

Definitions

¶1. A solution file is simply a list of commands which will win a work of IF, starting from turn 1. In this section we will generate this list given the Skein file for an Inform 7 project: to follow this code, it's useful first to read the "Walkthrough solutions" section of the "Releasing" chapter in the main Inform documentation.

We will need to parse the entire skein into a tree structure, in which each node (including leaves) is one of the following structures. We expect the Inform user to have annotated certain nodes with the text `***` (three asterisks); the solution file will ignore all paths in the skein which do not lead to one of these `***` nodes. The surviving nodes, those in lines which do lead to `***` endings, are called "relevant".

Some knots have "branch descriptions", others do not. These are the options where choices have to be made. The `branch_parent` and `branch_count` fields are used to keep these labels: see below.

```
define MAX_NODE_ID_LENGTH 32
define MAX_COMMAND_LENGTH 128
define MAX_ANNOTATION_LENGTH 128

typedef struct skein_node {
    char id[MAX_NODE_ID_LENGTH];                    uniquely identifying ID used within the Skein file
    char command[MAX_COMMAND_LENGTH];                text of the command at this node
    char annotation[MAX_ANNOTATION_LENGTH];          text of any annotation added by the user
    int relevant;                                    is this node within one of the "relevant" lines in the skein?
    struct skein_node *branch_parent;                the trunk of the branch description, if any, is this way
    int branch_count;                                the leaf of the branch description, if any, is this number
    struct skein_node *parent;                       within the Skein tree: NULL for the root only
    struct skein_node *child;                        within the Skein tree: NULL if a leaf
    struct skein_node *sibling;                      within the Skein tree: NULL if the final option from its parent
    MEMORY_MANAGEMENT
} skein_node;
```

The structure skein_node is private to this section.

¶2. The root of the Skein, representing the start position before any command is typed, lives here:

```
skein_node *root_skn = NULL;
```
only `NULL` *when the tree is empty*

---

§**1.** This section provides just one function to the rest of `cblorb`: this one, which implements the Blurb `solution` command.

Our method works in four steps. Steps 1 to 3 have a running time of $O(K^2)$, where K is the number of knots in the Skein, and step 4 is $O(K \log_2(K))$, so the process as a whole is $O(K^2)$.

```
void walkthrough(char *Skein_filename, char *walkthrough_filename) {
    build_skein_tree(Skein_filename);
    if (root_skn == NULL) {
        error("there appear to be no threads in the Skein");
        return;
    }
    identify_relevant_lines();
    if (root_skn->relevant == FALSE) {
        error("no threads in the Skein have been marked '***'");
        return;
    }
    prune_irrelevant_lines();
    write_solution_file(walkthrough_filename);
}
```

The function walkthrough is called from 3/rel.

§2. Step 1: building the Skein tree.

```
skein_node *current_skein_node = NULL;
void build_skein_tree(char *Skein_filename) {
    root_skn = NULL;
    current_skein_node = NULL;
    file_read(Skein_filename, "can't open skein file", FALSE, read_skein_pass_1, 0);
    current_skein_node = NULL;
    file_read(Skein_filename, "can't open skein file", FALSE, read_skein_pass_2, 0);
}
void read_skein_pass_1(char *line, text_file_position *tfp) { read_skein_line(line, 1); }
void read_skein_pass_2(char *line, text_file_position *tfp) { read_skein_line(line, 2); }
```

The function build_skein_tree is.
The function read_skein_pass_1 is.
The function read_skein_pass_2 is.

§**3.** The Skein is stored as an XML file. Its format was devised by Andrew Hunter in the early days of the Inform user interface for Mac OS X, and this was then adopted by the user interface on other platforms, so that projects could be freely exchanged between users regardless of their platforms. That makes it a kind of standard, but it isn't at present a public or documented one. We shall therefore make few assumptions about it.

```
void read_skein_line(char *line, int pass) {
    char node_id[MAX_NODE_ID_LENGTH];
    find_node_ID_in_tag(line, "item", node_id, MAX_NODE_ID_LENGTH, TRUE);
    if (pass == 1) {
        if (node_id[0]) ⟨Create a new skein tree node with this node ID 4⟩;
        if (current_skein_node) {
            ⟨Look for a "command" tag and set the command text from it 6⟩;
            ⟨Look for an "annotation" tag and set the annotation text from it 7⟩;
        }
    } else {
        if (node_id[0]) current_skein_node = find_node_with_ID(node_id);
        if (current_skein_node) {
            char child_node_id[MAX_NODE_ID_LENGTH];
            find_node_ID_in_tag(line, "child", child_node_id, MAX_NODE_ID_LENGTH, TRUE);
            if (child_node_id[0]) {
                skein_node *new_child = find_node_with_ID(child_node_id);
                if (new_child == NULL) {
                    error("the skein file is malformed (B)");
                    return;
                }
                ⟨Make the parent-child relationship 5⟩;
            }
        }
    }
}
```

The function read_skein_line is.

§**4.** Note that the root is the first knot in the Skein file.

⟨Create a new skein tree node with this node ID 4⟩ ≡

```
    current_skein_node = CREATE(skein_node);
    if (root_skn == NULL) root_skn = current_skein_node;
    strcpy(current_skein_node->id, node_id);
    strcpy(current_skein_node->command, "");
    strcpy(current_skein_node->annotation, "");
    current_skein_node->branch_count = -1;
    current_skein_node->branch_parent = NULL;
    current_skein_node->parent = NULL;
    current_skein_node->child = NULL;
    current_skein_node->sibling = NULL;
    current_skein_node->relevant = FALSE;
    if (trace_mode) printf("Creating knot with ID '%s'\n", node_id);
```

This code is used in §3.

§**5.** We make `new_child` the youngest child of `current_skein_mode`:

⟨Make the parent-child relationship 5⟩ ≡
```
new_child->parent = current_skein_node;
if (current_skein_node->child == NULL) {
    current_skein_node->child = new_child;
} else {
    skein_node *familial = current_skein_node->child;
    while (familial->sibling) familial = familial->sibling;
    familial->sibling = new_child;
}
```

This code is used in §3.

§**6.**

⟨Look for a "command" tag and set the command text from it 6⟩ ≡
```
char *p = current_skein_node->command;
if (find_text_of_tag(line, "command", p, MAX_COMMAND_LENGTH, FALSE)) {
    if (trace_mode) printf("Raw command '%s'\n", p);
    undo_XML_escapes_in_string(p);
    convert_string_to_upper_case(p);
    if (trace_mode) printf("Processed command '%s'\n", p);
}
```

This code is used in §3.

§**7.**

⟨Look for an "annotation" tag and set the annotation text from it 7⟩ ≡
```
char *p = current_skein_node->annotation;
if (find_text_of_tag(line, "annotation", p, MAX_ANNOTATION_LENGTH, FALSE)) {
    if (trace_mode) printf("Raw annotation '%s'\n", p);
    undo_XML_escapes_in_string(p);
    if (trace_mode) printf("Processed annotation '%s'\n", p);
}
```

This code is used in §3.

§**8.** Try to find a node ID element attached to a particular tag on the line:

```
int find_node_ID_in_tag(char *line, char *tag,
    char *write_to, int max_length, int abort_not_trim) {
    char portion1[MAX_TEXT_FILE_LINE_LENGTH], portion2[MAX_TEXT_FILE_LINE_LENGTH];
    char prototype[64];
    strcpy(prototype, "%[^<]<");
    strcat(prototype, tag);
    strcat(prototype, " nodeId=\"%[^\"]\"");
    write_to[0] = 0;
    if (sscanf(line, prototype, portion1, portion2) == 2) {
        if ((strlen(portion2) >= max_length-1) && (abort_not_trim)) {
            error("the skein file is malformed (C)");
            return FALSE;
        }
        strncpy(write_to, portion2, max_length-1); write_to[max_length-1] = 0;
        return TRUE;
    }
    return FALSE;
}
```

The function find_node_ID_in_tag is.

§**9.** Try to find the text of a particular tag on the line:

```
int find_text_of_tag(char *line, char *tag,
    char *write_to, int max_length, int abort_not_trim) {
    char portion1[MAX_TEXT_FILE_LINE_LENGTH], portion2[MAX_TEXT_FILE_LINE_LENGTH],
        portion3[MAX_TEXT_FILE_LINE_LENGTH];
    char prototype[64];
    strcpy(prototype, "%[^>]>%[^<]</");
    strcat(prototype, tag);
    strcat(prototype, "%s");
    if (sscanf(line, prototype, portion1, portion2, portion3) == 3) {
        if ((strlen(portion2) >= max_length-1) && (abort_not_trim)) {
            error("the skein file is malformed (C)");
            return FALSE;
        }
        strncpy(write_to, portion2, max_length-1); write_to[max_length-1] = 0;
        if (trace_mode) printf("found %s = '%s'\n", tag, portion2);
        return TRUE;
    }
    return FALSE;
}
```

The function find_text_of_tag is.

§**10.** This is not very efficient, but:

```
skein_node *find_node_with_ID(char *id) {
    skein_node *skn;
    LOOP_OVER(skn, skein_node)
        if (strcmp(id, skn->id) == 0)
            return skn;
    return NULL;
}
```

The function find_node_with_ID is.

§**11.** Finally, we needed the following string hackery:

```
void convert_string_to_upper_case(char *p) {
    int i;
    for (i=0; p[i]; i++) p[i]=toupper(p[i]);
}
```

The function convert_string_to_upper_case is.

§**12.** and:

```
void undo_XML_escapes_in_string(char *p) {
    int i = 0, j = 0;
    while (p[i]) {
        if (p[i] == '&') {
            char xml_escape[16];
            int k=0;
            while ((p[i+k] != 0) && (p[i+k] != ';') && (k<14)) {
                xml_escape[k] = tolower(p[i+k]); k++;
            }
            xml_escape[k] = p[i+k]; k++; xml_escape[k] = 0;
            ⟨We have identified an XML escape 13⟩;
        }
        p[j++] = p[i++];
    }
    p[j++] = 0;
}
```

The function undo_XML_escapes_in_string is.

§**13.** Note that all other ampersand-escapes are passed through verbatim.

⟨We have identified an XML escape 13⟩ ≡

```
    char c = 0;
    if (strcmp(xml_escape, "&lt;") == 0) c = '<';
    if (strcmp(xml_escape, "&gt;") == 0) c = '>';
    if (strcmp(xml_escape, "&amp;") == 0) c = '&';
    if (strcmp(xml_escape, "&apos;") == 0) c = '\'';
    if (strcmp(xml_escape, "&quot;") == 0) c = '\"';
    if (c) { p[j++] = c; i += strlen(xml_escape); continue; }
```

This code is used in §12.

§14. Step 2: identify the relevant lines. We aim to show how to reach all knots in the Skein annotated with text which begins with three asterisks. (We trim those asterisks away from the annotation once we spot them: they have served their purpose.) A knot is "relevant" if and only if one of its (direct or indirect) children is marked with three asterisks in this way.

```
void identify_relevant_lines(void) {
    skein_node *skn;
    LOOP_OVER(skn, skein_node) {
        char *p = skn->annotation;
        if (trace_mode) printf("Knot %s is annotated '%s'\n", skn->id, p);
        if ((p[0] == '*') && (p[1] == '*') && (p[2] == '*')) {
            int i = 3, j; while (p[i] == ' ') i++;
            for (j=0; p[i]; i++) p[j++] = p[i]; p[j] = 0;
            skein_node *knot;
            for (knot = skn; knot; knot = knot->parent) {
                knot->relevant = TRUE;
                if (trace_mode) printf("Knot %s is relevant\n", knot->id);
            }
        }
    }
}
```

The function identify_relevant_lines is.

§15. Step 3: pruning irrelevant lines out of the tree. When the loop below concludes, the relevant nodes are exactly those in the component of the tree root, because:

(a) No irrelevant node can be the child of a relevant one; and no relevant node can be the child of an irrelevant one by definition. So the tree falls into components each of which is fully relevant or fully not.
(b) Since we never break any relevant-parent-relevant-child relationships, the number of components containing at least one relevant node is unchanged.
(c) Since the Skein is initially a tree and not a forest, we start with just one component, and it contains the tree root, which is known to be relevant (we would have given up with an error message if not).
(d) And therefore at the end of the loop the "tree" consists of a single component headed by the tree root and containing all of the relevant nodes, together with any number of other components each of which contains only irrelevant ones.

```
void prune_irrelevant_lines(void) {
    skein_node *skn;
    LOOP_OVER(skn, skein_node)
        if ((skn->relevant == FALSE) && (skn->parent))
            ⟨Delete this node from its parent 16⟩;
}
```

The function prune_irrelevant_lines is.

§16.

⟨Delete this node from its parent 16⟩ ≡
```
if (skn->parent->child == skn) {
    skn->parent->child = skn->sibling;
} else {
    skein_node *skn2 = skn->parent->child;
    while ((skn2) && (skn2->sibling != skn)) skn2 = skn2->sibling;
    if ((skn2) && (skn2->sibling == skn)) skn2->sibling = skn->sibling;
}
skn->parent = NULL;
skn->sibling = NULL;
```
This code is used in §15.

§17. Step 4: writing the solution file.

```
void write_solution_file(char *walkthrough_filename) {
    FILE *SOL = fopen(walkthrough_filename, "w");
    if (SOL == NULL)
        fatal_fs("unable to open destination for solution text file",
            walkthrough_filename);
    fprintf(SOL, "Solution to \""); copy_placeholder_to("TITLE", SOL);
    fprintf(SOL, "\" by "); copy_placeholder_to("AUTHOR", SOL); fprintf(SOL, "\n\n");
    recursively_solve(SOL, root_skn, NULL);
    fclose(SOL);
}
```
The function write_solution_file is.

§18. The following prints commands to the solution file from the position `skn` – which means just after typing its command – with the aim of reaching all relevant endings we can get to from there.

```
void recursively_solve(FILE *SOL, skein_node *skn, skein_node *last_branch) {
    ⟨Follow the skein down until we reach a divergence, if we do 19⟩;
    ⟨Print the various alternatives from this knot where the threads diverge 20⟩;
    ⟨Show the solutions down each of these alternative lines in turn 21⟩;
}
```
The function recursively_solve is.

§19. If there's only a single option from here, we could print it and then call `recursively_solve` down from it. That would make the code shorter and clearer, perhaps, but it would clobber the C stack: our recursion depth might be into the tens of thousands on long solution files. So we tail-recurse instead of calling ourselves, so to speak, and just run down the thread until we reach a choice. (If we never do reach a choice, we can return – there is nowhere else to reach.)

⟨Follow the skein down until we reach a divergence, if we do 19⟩ ≡
```
while ((skn->child == NULL) || (skn->child->sibling == NULL)) {
    if (skn->child == NULL) return;
    if (skn->child->sibling == NULL) {
        skn = skn->child;
        write_command(SOL, skn, NORMAL_COMMAND);
    }
}
```
This code is used in §18.

§**20.** Thus we are here only when there are at least two alternative commands we might use from position `skn`.

⟨Print the various alternatives from this knot where the threads diverge 20⟩ ≡

```
fprintf(SOL, "Choice:\n");
int branch_counter = 1;
skein_node *option;
for (option = skn->child; option; option = option->sibling)
    if (option->child == NULL) {
        write_command(SOL, option, BRANCH_TO_END_COMMAND);
    } else {
        option->branch_count = branch_counter++;
        option->branch_parent = last_branch;
        write_command(SOL, option, BRANCH_TO_LINE_COMMAND);
    }
```

This code is used in §18.

§**21.**

⟨Show the solutions down each of these alternative lines in turn 21⟩ ≡

```
skein_node *option;
for (option = skn->child; option; option = option->sibling)
    if (option->child) {
        fprintf(SOL, "\nBranch (");
        write_branch_name(SOL, option);
        fprintf(SOL, ")\n");
        recursively_solve(SOL, option, option);
    }
```

This code is used in §18.

§22. Writing individual commands and branch descriptions.

```
define NORMAL_COMMAND 1
define BRANCH_TO_END_COMMAND 2
define BRANCH_TO_LINE_COMMAND 3

void write_command(FILE *SOL, skein_node *cmd_skn, int form) {
    if (form != NORMAL_COMMAND) fprintf(SOL, "  ");
    fprintf(SOL, "%s", cmd_skn->command);
    if (form != NORMAL_COMMAND) {
        fprintf(SOL, " -> ");
        if (form == BRANCH_TO_LINE_COMMAND) {
            fprintf(SOL, "go to branch (");
            write_branch_name(SOL, cmd_skn);
            fprintf(SOL, ")");
        }
        else fprintf(SOL, "end");
    }
    if (cmd_skn->annotation[0]) fprintf(SOL, " ... %s", cmd_skn->annotation);
    fprintf(SOL, "\n");
}
```

The function write_command is.

§**23.** For instance, at the third option from a thread which ran back to being the second option from a thread which ran back to being the seventh option from the original position, the following would print "7.2.3". Note that only the knots representing the positions after commands which make a choice are labelled in this way.

```
void write_branch_name(FILE *SOL, skein_node *skn) {
    if (skn->branch_parent) {
        write_branch_name(SOL, skn->branch_parent);
        fprintf(SOL, ".");
    }
    fprintf(SOL, "%d", skn->branch_count);
}
```

The function write_branch_name is.